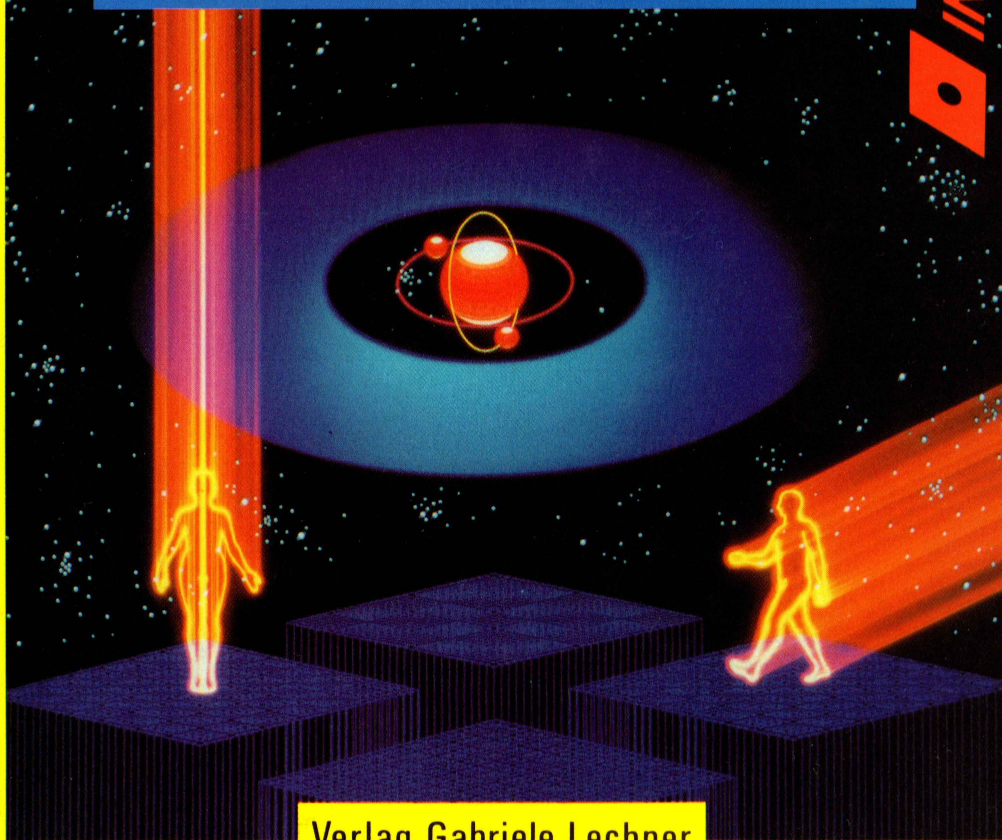
Niki Laber

Amiga Spiele selber programmieren

INKL. 1 DISK



Verlag Gabriele Lechner

SPIELE SELBER PROGRAMMIEREN

SPIELE SELBER PROGRAMMIEREN

Niki Laber

Verlag Gabriele Lechner

ISBN 3-926858-31-1
1. Auflage 1991

Autor : Niki Laber
Illustrationen : Niki Laber
Einbandgestaltung : Young Su Niedermeier
Titelfoto : Zefa ORION PRESS 2: J 8947
Druck : WB-Druck GmbH & Co.
Rieden am Forggensee

INHALT

VORWORT

In diesem Buch soll der Leser mit der Programmierung von Spielen vertraut gemacht werden. Da es sich hierbei um ein sehr komplexes Thema handelt, ist es im Rahmen dieses Buches nicht möglich, bei Grundlagen zu beginnen. Daher sollte der Leser folgende Voraussetzungen erfüllen:

1. Vertraut sein mit CLI und Workbench.

2. Kenntnisse über mathematische Grundlagen (Bit/Byte-Operationen, Boolsche Algebra) haben.

3. 68000 Assembler zumindest in den Grundzügen beherrschen (wobei dies nicht unbedingt für Hardware-Erfahrung gilt!).

4. Über die Bedienung eines Assemblers informiert sein (Seka Assembler wäre vorteilhaft).

Auf diese Punkte wird im Laufe des Buches nicht näher eingegangen. Literatur zu diesen Themen gibt es inzwischen genug (siehe Literaturverzeichnis).

In diesem Buch wird die Programmierung von Spielen ausschließlich in Assembler besprochen, da sich diese Sprache am besten dafür eignet. Andere Sprachen sind sicherlich einfacher, aber nur in Assembler hat man die optimale "Bewegungsfreiheit", um kurze und schnelle Programme zu schreiben, was besonders für Spiele sehr wichtig ist.

Die Programmierung eines Spieles bringt einige Probleme und viel Arbeit mit sich. Das fängt schon bei der Planung an. Viele Programmierer denken über dieses Problem nicht sorgfältig nach. Meistens hat man eine gute Idee für einen Spezial-Effekt oder eine extrem schnelle Routine und anschließend programmiert man daraus ein Spiel.

In den meisten Fällen führt diese Methode nicht zu einem befriedigenden Ergebnis. Gute Spiele bedürfen sorgfältiger Planung. Diesem entscheidenden Schritt ist ein eigenes Kapitel gewidmet.
Ein weiterer wichtiger Punkt ist die möglichst optimale Programmierung und Verkürzung der Programme. Das Schöne an Assembler ist die Vielfalt des Befehlssatzes. Der PRINT-Befehl in Basic ist immer gleich, das heißt er kann vom Benützer nicht beeinflußt werden. In Assembler hingegen kann man das Gleiche "kurz" oder "lang" programmieren. Dieses Thema wird ebenfalls ausführlich besprochen.

Natürlich folgt den theoretischen Grundlagen der größte Teil des Buches: Die Programmierung. Von der Bilddarstellung bis zur Joystickabfrage wird alles besprochen, was zur Spieleprogrammierung notwendig ist.

Ebenso finden Sie im Anhang die für Hardware-Programmierer wichtigen Register auf einen Blick.

Aus den oben genannten Zutaten werden wir gemeinsam Schritt für Schritt ein Spiel programmieren. Um Ihnen das Abtippen zu ersparen, ist dem Buch eine Diskette beigelegt, auf der Sie nicht nur alle Listings, sondern auch Grafiken und Musik finden.

Ich hoffe, daß Sie mit diesem Buch viel Freude haben, und schon bald Ihre ersten Spiele entwickeln.

Niki Laber — Wien, im August 1991

KAPITEL 1

DIE ERSTEN SCHRITTE

Zuerst entsteht eine Idee. Diese aber optimal in ein Programm umzusetzen ist nicht immer leicht. Daher befaßt sich dieses Kapitel mit der Ausarbeitung einer Idee und der Erstellung eines Konzeptes, nach welchem programmiert werden kann.

1.1. BETRIEBSSYSTEM JA ODER NEIN?

Ganz zu Anfang dieses Buches möchte ich eine Grundsatzfrage klären, die sehr oft unter Amiga-Besitzern diskutiert wird. Es handelt sich hierbei um das heißumkämpfte Betriebssystem des Amiga.

Die Workbench und damit das gesamte Betriebssystem haben mitgeholfen, daß der Amiga so weit verbreitet ist. Seine großartige anwenderfreundliche Benutzeroberfläche hat vielen Anfängern den Einstieg in die Computerwelt erleichtert. Da die Workbench integrierender Bestandteil des Betriebssystems ist, kann man jederzeit, vorallem mit Anwender-Software, praktisch und schnell arbeiten, ohne vorher umständliche Maustreiber-Programme oder ähnliches (siehe PC) zu laden.

Allerdings man muß auch berücksichtigen, daß der Amiga ursprünglich als Spielkonsole konzipiert war und daher mit speziell dafür gedachten Chips (z.B. Blitter,...) ausgestattet wurde. Als Commodore den Amiga aufkaufte, wurden diese Pläne verworfen und eine Mischung aus Heimcomputer und Spielkonsole entwickelt. Die vorhandenen Grafikchips setzte man für das Betriebssystem ein. Die Entwickler dieses Traumcomputers hatten eine genaue Vorstellung über dessen Funktionsweise. Das Ergebnis sollte ein Computer sein, der beiden Anforderungen gerecht wird.

Diese beiden Bereiche lassen sich jedoch nur schwer in einem Gerät realisieren. Denn will man eine Spielkonsole, so ist jegliches Betriebssystem störend. Bei einem Arbeitsprogramm ist hingegen Grafik eher überflüssig. Daher mußte es früher oder später zu einem Konflikt unter den Programmierern kommen.

Die Einen befürworten eine systemgerechte Programmierung, damit jedes Programm mit der Workbench lauffähig ist. Die Anderen hingegen bestehen auf unabhängig programmierbare Routinen, die ohne Betriebssystem auskommen.

Welche Methode ist die bessere? Diese Frage läßt sich nicht so einfach beantworten; Sehen wir uns die Vor- und Nachteile objektiv an:

Wird ein Programm systemgerecht programmiert, kann man jegliche Vorteile, wie zum Beispiel das Multitasking ausnützen und somit mehrere Programme gleichzeitig verwenden. Darüberhinaus kann jedes Programm auf der Festplatte installiert werden, wodurch sich bei längeren und aufwendigeren Programmen die Ladezeiten erheblich verkürzen. Hält sich der Programmierer an die Richtlinien von Commodore, so ist sein Programm auch auf allen neuen Betriebssystem-Versionen lauffähig.

Der Vorteil der direkten Hardware-Programmierung ist die Geschwindigkeit, die man dabei gewinnen kann, da man jede benötigte Routine hardwareorientiert schreiben muß. Außerdem muß man den Speicher selbst einteilen und bestimmen, wo eventuelle Programmteile geladen werden.

Dafür wird man in keinster Weise durch das Betriebssystem eingeschränkt. Der größte Nachteil ist wohl die Geschwindigkeit mit der das Betriebssystem arbeitet. Da das ursprüngliche System nicht in Assembler programmiert wurde, mußte man bis zur Version 2.0 auch dadurch Geschwindigkeitseinbußen hinnehmen. Das Betriebssystem verbraucht wegen seines großen Umfangs sehr viel Speicherplatz.

Es wird oft behauptet, daß bei den meisten Spielen der Geschwindigkeitsunterschied so gering ist, daß das Betriebssystem ebenfalls noch Platz hätte. Das ist oft richtig, aber wenn man genau nachdenkt, muß man sich wirklich fragen, welchen Vorteil es bringt, das System gleichzeitig zu aktivieren. Theoretisch könnte man spielen, während im Hintergrund ein Listing ausgedruckt wird. Doch wer macht das wirklich? Selbst wenn dies einmal vorkommen sollte, müßte das Spiel deswegen Einbußen hinnehmen. Daher verzichtet man in den meisten Fällen bei Spielen auf das Betriebssystem.

Denn wer spielen möchte, will ein möglichst schnelles, gut programmiertes Spiel. Wenn man hingegen arbeiten möchte, wird man kaum, z.B. neben der Textverarbeitung auf ein Spiel umschalten.

Die direkte Hardware-Programmierung erfordert viel Wissen, damit man Spiele auch für zukünftige Geräte kompatibel erstellen kann. Denn man hat zwar die Möglichkeit sich frei im Computerspeicher zu bewegen, sollte aber dennoch einige Regeln beachten, sonst passiert es, wie man jetzt sehr oft sieht, daß Software nicht mehr korrekt läuft (z.B. Amiga 3000).

Da wir in diesem Buch die Hardware behandeln, werden wir für unsere Programme das Betriebssystem trotz seiner Vorteile nicht verwenden, sondern alle benötigten Routinen selber entwickeln. Dabei soll auch darauf geachtet werden, daß die Kompatibilität gewahrt bleibt, das heißt, unser Spiel soll auch auf dem Amiga 3000 lauffähig sein.

1.2. EINE IDEE WIRD GEBOREN

Das wohl schwierigste Problem, das sich einem Programmierer, der ein Spiel schreiben will, stellt, ist die "gute Idee". Denn ein wirklich interessantes Spiel kann nur entstehen, wenn mehrere Voraussetzungen erfüllt sind.

- eine gute Idee
- ein Konzept
- genaue Programmierung
- schöne Grafik
- gute Musik, Sound

Von diesen Komponenten hängt das Spiel ab. Stimmt nur einer dieser Punkte nicht, so kann sich dies negativ auf das gesamte Spiel auswirken. Denn was nützt die schönste Grafik, wenn das Spielprinzip total veraltet ist.

Will man alle Punkte beachten und genau ausführen, so ist das für einen einzelnen ziemlich schwer. Außerdem würde es wahrscheinlich lange dauern. Daher bilden sich zunehmend Programmier-Teams, wobei jedem einzelnen Mitglied eine andere Aufgabe zugeteilt wird. Einer programmiert, ein anderer zeichnet die Bilder und der nächste komponiert die Musik. Dadurch kann sich jeder auf ein bestimmtes Gebiet spezialisieren und es werden deutlich bessere Ergebnisse erzielt.

Nebenbei wird man auch schneller fertig, da parallel gearbeitet werden kann. Gut ist es, wenn man jemanden hat, der die Idee zum Spiel liefert und sowohl Planung als auch Konzept ausarbeitet. Eine weitere Rolle gewinnt immer mehr an Bedeutung, nämlich die des Geschichten-Schreibers. Denn ein gutes Spiel braucht mehr als eine kurze Spielanleitung, in der man aufgefordert wird eine Prinzessin zu befreien.

Wir wollen nun im Laufe dieses Buches alle Schritte von der Idee über Planung und Konzept bis hin zur Programmierung ausarbeiten, um letztendlich ein kleines Geschicklichkeitsspiel zu erstellen.

KAPITEL 2

DIE PLANUNGSPHASE

Da die Planung genauso wichtig, ja fast sogar noch wichtiger als die Programmierung ist, haben wir ihr ein eigenes Kapitel gewidmet. Ebenso finden Sie hier Tips zur optimalen Programmierung und zur Speicherbelegung.

2.1. WIE ERSTELLT MAN EIN KONZEPT

Um ein Konzept zu erstellen, sind mehrere Schritte erforderlich:

- Zuerst benötigt man eine, wie im vorhergehenden Kapitel schon besprochen. Diese sollte man in groben Zügen notieren, damit sie nicht vergessen wird.

- Anschließend sollte man versuchen, diese ersten Gedanken in ein 'Computerspiel gerechtes' Schema zu bringen. Das heißt, die Spielelemente müsen so umgewandelt bzw. angepaßt werden, daß sie für ein Computerspiel geeignet sind.

- Der nächste Schritt führt etwas weiter ins Detail. Alle Elemente, die das fertige Spiel enthalten soll, werden geordnet und zu Papier gebracht. Dazu gehört nicht nur das reine Spielgeschehen, sondern auch die Steuerung, die Bildschirmaufteilung, Ein- oder Zweispielermodus und ähnliches.

- Ist dies geschehen, wird das gesamte Spiel von der programmtechnischen Seite beleuchtet. Das heißt, es werden Details wie z.B. Bildschirmauflösung, Anzahl von Objekten (Bobs, oder reichen Sprites?), Scrolling, u.s.w. bestimmt und ebenfalls auf Papier festgehalten.

- Außerdem sollte man sich Gedanken darüber machen, wieviel Platz man diesem Spiel eingesteht. Verwendet man Grafik, Sound und Musik, so wird man vielleicht nicht mehr mit einer einzigen Diskette auskommen. Daher sollte man sich auch rechtzeitig nach geeigneten "Crunch"-Programmen umsehen, die das fertige Spiel komprimieren.

- Der letzte Schritt in der Reihe der Vorbereitungen, ist die Vergabe der einzelnen Aufgaben. Es heißt zwar immer, viele Köche verderben den Brei, aber bei größeren Spielprojekten ist dies nicht zutreffend. Je mehr Spezialisten daran arbeiten, desto besser ist das Endprodukt. Außerdem verkürzt sich die Zeit der Erstellung, da man parallel arbeiten kann.

Die aufgeführten Punkte sollten alle in schriftlicher Form vorliegen und an jeden Beteiligten (Programmier, Grafiker, Musiker, ...) ausgegeben werden, damit klare Richtlinien bestehen, an die sich jeder zu halten hat.

Ein weiterer wichtiger Punkt neben dem Konzept, ist der gesamte Ablauf von der Idee bis hin zum Vertrieb des fertigen Spiels:

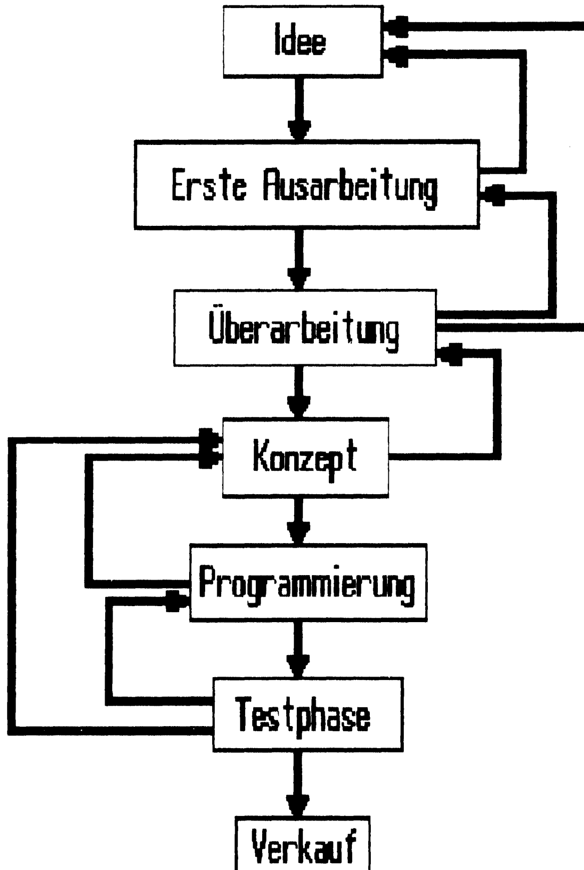


BILD 1: Der Planungsablauf

Nachdem das Konzept ausgearbeitet ist und die Aufgaben an die beteiligten Personen verteilt wurden, kann die Programmierung beginnen. Im Konzept sollte auch jemanden vorsehen sein, der quasi die Leitung, bzw. den Vorsitz übernimmt und alles koordiniert. Ebenso müssen von diesem die fertigen Einzelteile, wie Musik, Grafik, Sound und das Spiel selbst zusammengefügt werden. Danach folgt der oftmals, vorallem bei Simulationen, schwierigste Abschnitt, die Testphase. Während dieser Zeit sollten sämtlich Fehler beseitigt werden, bevor das fertige Produkt ausgeliefert werden kann.

Zunächst werden wir jetzt die obigen Schritte anhand unseres kleinen Spieles durchführen. Zuerst einige Worte zur Spielidee: Da man mit diesem Buch das Spieleprogrammieren erlernen soll, haben wir uns für ein kleines Geschicklichkeitsspiel entschieden. Dieses Programm ist nicht sehr umfangreich, damit es für Anfänger nicht zu verwirrend wird. Ebenso wurde versucht, möglichst viele Programmteile, wie z.B.: Scrolling, Coppereffekte, ... in das Spiel zu integrieren, das Ganze aber trotzdem leicht verständlich zu gestalten.

Aufgabenstellung:

Der Spieler muß mit seinem Spielsymbol, welches durch einen in Port 1 angesteckten Joystick gesteuert wird, auf einem Spielraster zwei verschiedenfarbige, bewegte Objekte hintereinander berühren. Der Spielraster soll aus zwei Ebenen bestehen, die vertikal in verschiedenen Geschwindigkeiten gescrollt (bewegt) werden. Erschwert wird dies dadurch, daß sich der Spieler nur über eine dieser Ebenen bewegen darf, da ihm sonst Lebensenergie abgezogen wird. Zwei weitere animierte Objekte bewegen sich auf diesen Ebenen.

Es ist nun die Aufgabe des Spielers, mit seinem Symbol zuerst das eine und anschließend das zweite Objekt zu berühren. Ist dies geschehen, soll an der im unteren Teil des Bildschirms befindlichen Tafel, eines von zehn Symbolen abgehakt werden. Sind zehn dieser Symbole durchgestrichen, so ist das Spiel zu Ende.

Damit das Spiel nicht zu einfach wird, ist noch ein Gegner eingebaut, der vom Spieler nicht berührt werden darf. Wenn doch, so verliert man eines seiner drei Bildschirmleben.

Sehen wir uns diese Beschreibung in grafischer Form an:

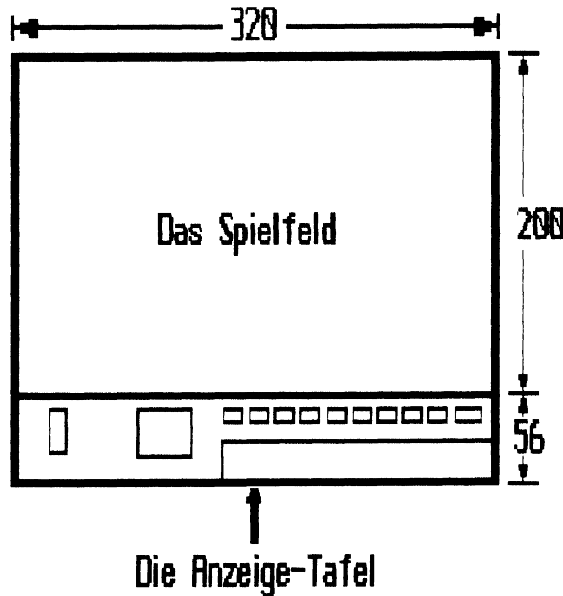


BILD 2: Das Spielfeld

- Das Spielfeld selbst besteht aus zwei, voneinander unabhängigen Playfields (Erklärungen dazu folgen im Kapitel Playfields), welche unterschiedlich groß sind und vertikal in verschiedenen Geschwindigkeiten gescrollt (bewegt) werden.

- Im Hintergrund soll als weitere Ebene eine, mittels Copper erzeugte, 3D-Rolle auf und ab bewegt werden.

- Das Spielfeld selbst ist nur 320x200 Bildpunkte groß. Der untere Teil (PAL) soll für eine Anzeigetafel genützt werden. Dazu müssen wir den Bildschirm aufteilen. Diese Tafel beinhaltet nicht nur die Anzahl der Bildschirmleben, sondern auch die bereits erfolgten Berührungen des Spielers, sowie dessen Lebensenergie.

- Das Symbol des Spielers wird durch einen, in Port 1 angesteckten, Joystick gesteuert.

- Die restlichen drei Symbole werden zufällig über den Bildschirm bewegt.

- Außerdem soll zusätzlich die linke Maustaste abgefragt und auf Druck derselben, das Spiel wieder beendet werden.

- Die Objekte werden selbstverständlich animiert.

- Ebenso soll ein Sample als Hintergrund-Musik abgespielt werden.

Nun haben wir das Konzept sowohl grafisch dargestellt, als auch in Worte gefaßt, damit wir es bei der Programmierung leichter haben. In jedem Kapitel werden wir auf die einzelnen Punkte näher eingehen.

2.2. DIE SPEICHERBELEGUNG

Sehr wichtig für die Spiele-Programmierung ist die Kenntnis des Speicherbelegung, denn wenn man Hardware programmiert, sollte man genau wissen, wo man Daten ablegen kann, wo Speichererweiterungen und das Kickstart-ROM liegen. Sehen wir uns daher folgende Belegung an:

```
$000000   ---------------------------------------
          |           (256 KB RAM A1000)         |
$040000   ----------- 512 KB Chip-RAM -----------
          | (256 KB Speichererweiterung A1000)   |
$080000   ---------------------------------------
          |              reserviert              |
$200000   ---------------------------------------
          |             8 MB Fast RAM            |
$a00000   ---------------------------------------
          | teilweise belegt von CIA Registern   |
$c00000   ---------------------------------------
          |    512 KB Speichererweiterungen      |
$c80000   ---------------------------------------
          |          nicht in Verwendung         |
$d00000   ---------------------------------------
          |  teilweise belegt von Customchips,   |
          |          Erweiterungen, u.ä.         |
$e00000   ---------------------------------------
          |              reserviert              |
$e80000   ---------------------------------------
          |            Expansionsport            |
$f00000   ---------------------------------------
          |              reserviert              |
$f80000   ---------------------------------------
          |    Spiegelung des Kickstart ROMs     |
$fc0000   ---------------------------------------
          |         256 KB Kickstart ROM         |
$ffffff   ---------------------------------------
```

Wie wir aus der vorigen Darstellung entnehmen können, ist der 68000er in der Lage, einen Speicherbereich von $000000 bis $ffffff zu adressieren. Dies entspricht einer Adreßbreite von 16 MegaByte. 2 hoch 24 (24 Bit breiter Adreßbus) ergibt 16,777216 MegaByte.

Der unterste Speicherbereich (von $000000 bis $080000) wird Chip-RAM genannt, da auf diesen Speicherbereich die Customchips Einfluß nehmen können. Der Zugriff dieser Chips auf höher liegende Speicherplätzen ist nicht möglich, denn der ist nur dem 68000er vorbehalten. Daher wird dieser Speicherbereich auch Fast-Mem genannt. Es gibt inzwischen die Möglichkeit 1 MB Chip-RAM zu verwenden.

Die Speicherbereiche $a00000 und $d00000 werden teilweise von den Customchips und CIA-Registern verwendet. Diese Register werden wir später noch genauer kennenlernen.

Im obersten Teil des Speichers befindet sich das Betriebssystem (ROM) und dessen Spiegelung.

2.3. OPTIMIERTE PROGRAMMIERUNG

Assembler wird zurecht als komplizierteste Computersprache bezeichnet. Denn der wohl größte Nachteil dieser Sprache ist die große Anzahl von Befehlen und die Vielfalt wie diese Befehle verwendet werden können. Dafür hat der Programmierer uneingeschränkte Freiheit beim Arbeiten. Er kann nicht nur über Speichereinteilung und Verwaltung selbst verfügen oder die letzten Tricks und Effekte aus den Chips kitzeln, sondern auch Einfluß auf die Geschwindigkeit seiner Programme nehmen. In anderen Sprachen, vorallem in den sogenannten höheren Programmiersprachen wie zum Beispiel in Basic, ist es nicht möglich einen Befehl schneller zu machen. Ein Beispiel: "PRINT" bleibt "PRINT" und wird nicht schneller. In Assembler hingegen kann man diesen Befehl "langsam" oder "schnell" programmieren. Wir haben an dieser Stelle einige Tricks aufgeführt, die mithelfen, Ihre Programme kürzer und schneller zu machen.

Register löschen

Das Einsparen sowohl von Taktzyklen als auch von Bytes beginnt schon mit dem Löschen von Registern. Die wohl umständlichste Art ein Register zu löschen ist:

```
move.l #$00000000,d5
```

Um ein Datenregister zu löschen, wird wohl kaum jemand diesen Befehl verwenden. Viel häufiger wird

```
clr.l d5
```

verwendet. Viel schneller, was viele jedoch nicht wissen ist das Kommando

```
moveq.l #0,d5
```

Dieser Befehl benötigt nur 2 Bytes und 4 Taktzyklen wogegen clr.l um 2 Taktzyklen mehr benötigt. Der clr Befehl hat den Vorteil, daß er auch Word- und Byteweise ausführbar ist.

Mit Adreß-Registern verhält es sich ähnlich. Aber die clr oder moveq Befehle gelten nicht für Adreß-Register. Dennoch sollte man

```
move.l #0,a0
```

vermeiden und statt dessen

```
suba.l a0,a0
```

verwenden. Denn wenn man von a0 dessen Inhalt subtrahiert, erhält man ebenfalls 0. Zudem ist die zweite Version um vier Byte kürzer und um 4 Zyklen schneller.

Quick-Befehle

Weitere Einsparungen lassen sich durch die Verwendung der schon erwähnten Quick-Befehle erzielen. Der Amiga bietet uns drei dieser Befehle an. Den ersten, moveq, haben wir bereits kennengelernt. Die anderen beiden sind subq und addq.

```
add.l #1,Speicherstelle

addq.l #1,Speicherstelle
```

Addq ist hier sogar um die Hälfte kürzer und spart 8 Taktzyklen. Der Nachteil ist, daß addq und subq nur eine drei Bit-Zahl (dies entspricht einer Zahl von 0-7) verarbeiten kann. Will man Zahlen größer als 7 addieren, so kann man den lea-Befehl verwenden. Dieser ist aber nur in Verbindung mit Adreß-Registern möglich. Statt

```
add.l #10,a0
```

verwendet man

```
lea 10(a0),a0
```

und spart dadurch zwar keine Bytes aber immerhin 6 Taktzyklen.

PC-relative Adressierung

Platz läßt sich auch sparen, wenn man die sogenannte PC-relative Adressierung verwendet. Was bedeutet nun PC-relativ? Wie Sie sicherlich wissen, ist der Amiga ein multitasking-fähiges Gerät. Das bedeutet, daß mehrere Programme scheinbar gleichzeitig verarbeitet werden können. Dadurch wird es nötig, daß jedes Programm an jeder beliebigen Adresse im Speicher funktionieren muß. Daher wird nun für jedes Programm ein Code erzeugt, der bei Adresse 0 beginnt. Im File ist eine Tabelle enthalten, die vor dem Start das Programm an die jeweilige Adresse anpaßt. Jede im Programmcode enthaltene 32-Bit Adresse enthält ein zusätzliches Longwort im sogenannten Relocation-Hunk. Programmiert man jedoch PC-relativ, entfällt nicht nur der Eintrag in diese Tabelle, sondern das Programm wird kürzer und sogar schneller. Sehen wir uns hierzu ein Beispiel an, statt

```
lea copperliste,a1
```

verwendet man die PC-relative Version:

```
lea copperliste(pc),a1
```

Der einzige Nachteil ist, daß dies nur bei einem Distanzwert einer vorzeichenbehafteten 16-Bit Zahl funktioniert. Das bedeutet in unserem Fall, daß der Label 'copperliste' nicht weiter als +32768 oder -32767 Bytes entfernt sein darf. Daher sollte man, wenn man PC-relativ programmiert, solche Labels in der Mitte seines Programms plazieren, damit sie sowohl von vorne, als auch von hinten erreichbar sind.
Leider ist der lea-Befehl nur bei Adreß-Registern zuläßig. Will man jedoch platzsparend Adressen in ein Datenregister einlesen, so empfiehlt es sich zu folgendem Trick zu greifen. Statt

```
move.l #copperliste,d0
```

nimmt man

```
lea    copperliste(pc),a0
move.l a0,d0
```

Es sind zwar zwei Befehle statt einem, aber trotzdem spart man immerhin vier Bytes. Von der Geschwindigkeit her sind beide Versionen gleich.

Die PC-relative Programmierung kann man ebenso bei Sprungbefehlen einsetzen. Um eine Unterroutine anzuspringen gibt es den Befehl

```
jsr unterroutine
```

Mit PC-relativem Code wird diese Routine natürlich kürzer:

```
jsr unterroutine(pc)
```

Die gleiche Adreßdistanz bietet uns aber auch der Befehl

```
bsr unterroutine
```

Daher ist dieser der obigen Version vorzuziehen.

Die Short-Befehle

Um die oben erklärten Sprungbefehle noch weiter zu verkürzen, sind an dieser Stelle die Short-Kommandos erwähnt. Beim 68000er haben wir die Möglichkeit, bei Verzweigungs-Befehlen, die im Byte-Bereich liegen (von -127 bis +128), den Befehl durch den Anhang

```
xxx.s
```

zu verkürzen. Daher lautet unser Sprungbefehl

```
bsr.s unterroutine
```

und ist somit um weitere 2 Byte kürzer. Selbstverständlich gibt es die Short Version bei allen anderen Sprung-Befehlen auch, wie zum Beispiel

```
beq.s label1

bne.s label2

blt.s label3

...
```

Weitere Adressierungsarten

Da Hardware-Programmierer die Register fast immer direkt ansprechen, kann man mit einem einfachen Trick auch einiges an Zeit einsparen. Will man zum Beispiel einige Farbwerte in die zugehörigen Farbregister eintragen so wird das oft so erledigt:

```
move #$f00,$dff180
move #$0f0,$dff182
...
```

Besser und kürzer ist es, wenn man die Basisadresse vorher in ein Register einträgt und die Farben über dieses in das Hardware-Register schreibt.

```
lea $dff180,a0
move #$f00,(a0)
move #$0f0,2(a0)
...
```

Noch schneller wird das Programm, wenn man folgende Kombination verwendet:

```
lea $dff180,a0
move #$f00,(a0)+
move #$0f0,(a0)+
...
```

Will man jedoch nur einen Farbwert eintragen, so ist natürlich die allererste Version die beste und schnellste, aber bereits ab zwei Befehlen ist die letzte von Vorteil. Es empfiehlt sich überhaupt, beim Hardware-Programmieren, die Basisadresse ($dff000), der Hardware-Register in ein Adreß-Register einzulesen.

```
lea $dff000,a6
```

Am besten verwendet man ein Adreß-Register, das man sonst eher nicht benötigt. Dadurch kann man immer auf die Basisadresse zugreifen. Allerdings muß man beachten, daß es ziemlich sicher zu einem Absturz kommen wird, wenn dieses Register überschrieben wird und man anschließend wieder auf Hardware-Register zugreifen will.

Darüberhinaus läßt sich noch Zeit und Speicherplatz sparen, wenn man keine unnötig "lange" Adressierung verwendet. Denn es gibt die sogenannte "absolut kurze" Adressierung. Will man zum Beispiel eine eigene Interrupt-Routine schreiben, so muß zuerst der originale Einsprung gerettet werden, dies wird von den meisten Programmierern so gelöst:

```
move.l $6c,saveirq
```

Viele wissen aber nicht, wie man die absolut kurze Adressierung richtig einsetzt. Mittels

```
move.l $6c.w,saveirq
```

werden nicht nur 2 Byte, sondern auch noch 4 Taktzyklen eingespart.

Multiplizieren und Dividieren

Der 68000er bietet für Multiplikationen und Divisionen die Befehle mulu, muls, divu und divs an. Für den Programmierer recht angenehm, da er zwischen vorzeichenrichtiger Multiplikation oder Division wählen kann. Leider sind diese Operationen aber extrem zeitintensiv, mulu/muls benötigen ungefähr 80 Taktzyklen. Divu/divs hingegen noch fast das Doppelte. Daher sollte man diese Befehle, wann immer es möglich ist, vermeiden. Bei einfachen Multiplikationen mit 2 ist der einfache add-Befehl vorzuziehen:

```
mulu #2,d6
```

ist viel langsamer als

```
add.l d6,d6
```

Denn bei Additionen mit 2 ist es viel kürzer und extrem schneller, wenn man den Inhalt des jeweiligen Registers zu sich selber hinzuaddiert.

Hat man es mit größeren Multiplikationen zu tun, kann man mit den lsl- und asl-Befehlen Abhilfe schaffen.

```
mulu #16,d6
```

wird durch den Gebrauch von

```
lsl.l #4,d6
```

nich nur um 4 Bytes kürzer, sondern auch um ungefähr 60 Taktzyklen schneller. Analog dazu ist bei divs/divu lsr und asr zu verwenden.

Longwords, Words oder Bytes

Eine weitere Methode um vorallem Speicherplatz zu sparen, ist die genaue und gezielte Verwendung von Longwords, Words und Bytes. Nur allzu oft werden unnötigerweise Longwords definiert, obwohl ein Byte völlig ausgereicht hätte. Ein Beispiel:

```
          move.l #15,d0
          move.l #145,d1
          add.l d0,d1
          move.l d1,speicher

speicher: dc.l 0
```

Übersteigt der Wert der Addition 255 nicht, so genügt folgendes Listing:

```
          move.b #15,d0
          move.b #145,d1
          add.b #d0,d1
          move.b d1,speicher

speicher: dc.b 0
```

Dieser Vorgang spart nur fast unwesentlich an Zeit, aber immens an Speicherplatz. Außerdem kann man Platz sparen, wenn man bei Eintreffen eines bestimmten Ereignisses, nicht ein ganzes Byte "blockiert", sondern sich mit einem Bit begnügt. Ein Byte besteht bekanntlich aus 8 Bit. Daher könnte man beispielsweise 8 verschiedene Ereignisse darin festhalten.

```
cmp.b #1,test
...
cmp.b #1,test1
...
cmp.b #1,test2
...
```

Anstatt einen bestimmten Wert abzufragen, könnte man sich mit einem Bit des Bytes "test" begnügen, somit sparen wir uns die Definition der anderen Bytes (test1, test2,...). Der Amiga stellt uns sogar einige Befehle für die Arbeit mit Bits zur Verfügung:

```
btst, bchg, bset und bclr
```

Wir sparen also Speicher, wenn wir diese Befehle einbauen.

```
btst #1,test
...
btst #2,test
...
btst #3,test
...
```

Wie man ein Register oder eine Speicherstelle möglichst schnell löscht, haben wir bereits gelernt. Aber wie ist es, wenn wir einen anderen Wert als 0 in einer Speicherstelle für eine spezielle Abfrage benötigen?

```
move.b #$ff,speicherstelle
```

Bei dem Wert $ff ist dies kein Problem, da hierfür ein bestimmter Befehl existiert, der viel schneller ist:

```
st speicherstelle
```

Dieser Befehl schreibt den Wert $ff in die angegebene Speicherstelle.

Kürzere Schleifen

Will man eine Schleife programmieren, so sollte man dazu möglichst nicht die Kommandos sub/cmp/bne verwenden, sondern mit den vom 68000er zur Verfügung gestellten dbcc-Schleifen arbeiten.
Ein Beispiel zu Verdeutlichung: Es sollen 100 Longwords von a0 nach a1 kopiert werden.

```
schleife:   move.l (a0)+,(a1)+
            add.l #1,d0
            cmp.l #100,d0
            bne  schleife
```

Die oben gezeigte Schleife ist besonders schlecht programmiert, nicht nur, weil die dbcc-Schleife keine Anwendung gefunden hat, sondern weil unnötigerweise auch mit Longwords und ohne Short-Anweisung gearbeitet wurde. Zum Vergleich eine etwas bessere Version:

```
schleife:   move.l (a0)+,(a1)+
            add.b #1,d0
            cmp.b #100,d0
            bne.s schleife
```

Aber nun folgt die beste Möglichkeit:

```
            move #99,d0
schleife:   move.l (a0)+,(a1)+
            dbra d0,schleife
```

Mit diesem Programm wurden nicht nur Byte sowohl im Code als auch im Quelltext gespart, sondern auch ganz erheblich die Geschwindigkeit gesteigert. Die dbcc-Befehle sind mit dbeq oder dbne möglich. Man sollte diese Kommandos den cmp-Befehlen vorziehen und versuchen, seine Programme anzupassen, sprich anstatt einen Zähler zu erhöhen, einfach mittels dbcc auf -1 zählen lassen. Ist trotzdem einmal ein Vergleich mit 0 nötig,

```
        cmp.b #0,speicherstelle
        ...
```

spart man auch, wenn man den tst-Befehl verwendet:

```
        tst.b speicherstelle
        ...
```

Dieser Befehl existiert sowohl im Byte-, Word- als auch im Longword-Format.

Manchmal können Schleifen trotz dbcc-Anweisung viel zu lange dauern. Daher ist man schneller, wenn man die Befehle direkt hintereinander in den Speicher schreibt. Diese Methode hat den Nachteil, daß sie ziemlich speicheraufwendig ist. Es liegt nun am Programmierer, zu entscheiden, was ihm wichtiger ist, Speicher oder Geschwindigkeit.

```
           move #9,d0
schleife:  clr.l (a0)+
           dbra d0,schleife


           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
           clr.l (a0)+
```

Sie sehen sofort, wie Speicherintensiv die zweite Version ist, dafür ist sie unheimlich schnell, was manchmal ausschlaggebend sein kann.

Dies sind nur einige Tips und Tricks, die mithelfen ein Programm kurz und zeitsparend zu schreiben. Sie werden merken, daß dies gar nicht so einfach ist. Man muß schon ein erfahrener Programmierer sein, um möglichst viele dieser Tips zu beherzigen. Selbst wenn man diese Methoden kennt, ist man oft schlampig oder findet es nicht der Mühe Wert seine Programme zu verkürzen. Man sollte aber schon von Beginn an versuchen korrekt zu arbeiten, auch wenn man glaubt, keine Probleme mit den Taktzyklen zu bekommen; denn gerade bei Spielen hat man diese Sorgen öfter als man denkt. Besonders bei längeren Programmen ist sehr schwer eine bereits fertige Routine nachträglich zu optimieren.

KAPITEL 3

DIE PROGRAMMIERUNG

In diesem Kapitel wird auf die Programmierung des Amiga, insbesondere der Custom-Chips eingegangen. Beginnend beim Copper, der für die Bilddarstellung verantwortlich ist, über Playfields und Sprites bis hin zum Blitter wird alles genau beschrieben und in ein Spiel eingebaut.

3.1. DER COPPER

In den vorhergehenden Kapitel, wurden die Entstehung eines Computerspieles und dessen theoretische Programmierung besprochen, sowie einige Beispiele für die Optimierung der Programme gegeben. In den nun nachfolgenden Kapitel wollen wir die praktische Seite näher betrachten.

Neben dem 68000er gibt es noch weitere Zusatz-Chips, die dem Prozessor Arbeit abnehmen sollen, damit dieser entlastet wird. Der erste, den wir hier besprechen wollen, der auch eine wichtige Rolle bei der Bilddarstellung spielt, ist der Copper (Coprozessor). Dieser Prozessor hat die Aufgabe, Werte in Registern an beliebiger Bildschirmposition zu verändern. Um dies genauer zu erklären, ist es nötig etwas weiter auszuholen:

Um ein Bild auf einem Fernseher oder Monitor darzustellen, verwendet man elektrisch geheizte Kathodenstrahlröhren. An deren Ende treten negative Elektronen aus, die gebündelt und abgelenkt werden. Der dadurch entstehende Strahl trifft auf einer flureszierenden Schicht auf und bringt diese zum Leuchten. Durch magnetische Felder wird dieser Strahl von links nach rechts und Zeile für Zeile von oben nach unten gelenkt. Dadurch entsteht ein sichtbares Schwarz/Weiß-Bild. Will man hingegen ein farbiges Bild erhalten, so kommt die additive Farbmischung zur Anwendung. Man verwendet statt einer Kathodenstrahlröhre gleich drei und läßt sie auf ein, jeweils mit roter, grüner oder blauer Farbe beschichtetem, Material treffen. Ist diese beschichtete Oberfläche genügend fein gerastert, werden die einzelnen Pixel vom Auge als ein Punkt erfaßt, und man sieht die gewünschte Mischfarbe. Durch diese Farbmischung können alle Farben gebildet werden. Bei maximal eingestellten Strahlenströmen entsteht die Farbe weiß, bei minimalen Strömen (= null) hingegen entsteht schwarz. Damit das Auge den Bildschirmaufbau nicht mehr wahrnimmt, muß dies ziemlich schnell geschehen. Bei einer Netzfrequenz von 50 Hertz wird das Bild durch den Elektronenstrahl eben 50 Mal aufgebaut.

Nun, was hat dies mit dem Copper zu tun? Der Copper ist, wie oben schon erwähnt, in der Lage, an jeder beliebigen Position, Register zu verändern. Das bedeutet, er kann z.B. die Auflösung mitten im Bild umschalten. Dadurch kann beispielsweise der obere Teil in Lo-res und der untere Teil in Hi-res angezeigt werden. Dies kommt auch auf der Workbench zur Anwendung, wenn man mehrere Screens übereinander legt. Natürlich war dies noch lange nicht alles, sondern es ergeben sich unzählige Möglichkeiten und Tricks, mit denen man vorallem für Spiele schöne Effekte zaubern kann.

3.1.1. DIE COPPER-BEFEHLE

Der Copper besitzt genauso wie der 68000er einen Befehlssatz, also eine Reihe von Befehlen, mit denen der Prozessor gesteuert wird. Der Befehlssatz des 68000er ist im Gegensatz zum Copper riesig, da dieser nur drei Befehle kennt. Diese drei sind der Move-, Wait- und Skipbefehl.

Der MOVE-Befehl

Dieser Befehl ermöglicht es dem Copper, Customchip-Register zu beschreiben. Wie alle Copper-Kommandos besteht auch der Move-Befehl aus zwei Befehlsworten. Im ersten Wort steht die Customchip Adresse, die verändert werden soll und im zweiten steht der Wert, mit dem dieses Register beschrieben werden soll. Als Kennzeichen für den Move-Befehl dient das gelöschte Bit 0 des ersten Befehlswortes. Da alle Customchip-Register sowieso an geraden Adressen liegen, ist dieses Bit automatisch immer gelöscht.

Normalerweise kann man mit dem Copper die Register von $0000 bis $007f nicht beeinflußen. Wird aber das unterste Bit im sogenannten COPCON-Register ($dff02e) gesetzt, so wird es für den Copper möglich die Register $0040 bis $007f zu verwenden und somit auch den Blitter zu beeinflußen. Der Zugriff auf die untersten Register von $0000 bis $0040 ist allerdings nicht möglich.

Ein Beispiel:

```
dc.w $0180,$0fff
```

Dieser Befehl schreibt in das COLOR00-Register den Wert $0fff ($0fff entspricht der Farbe weiß).

Der WAIT-Befehl

Dieser Befehl veranlaßt den Copper so lange zu warten, bis der Elektronenstrahl eine gewisse Position erreicht hat. Diese kann sowohl vertikal, als auch horizontal sein. Als Kennung ist Bit 0 immer auf 1 zu setzten. Die Bits 1-7 geben die horizontale, die Bits 8-15 die vertikale Position an. Im zweiten Befehlswort stehen die Maskenbits. Diese bestimmen, welche Bits der horizontalen und vertikalen Position überhaupt zum Vergleich mit der Rasterstrahlposition verwendet werden sollen. Die Maskenbits erstrecken sich über die Bits 1-14. Bit 15 wird auch das Blitter Finish Disable Bit genannt. Dies dient bei Verwendung von Blitter durch den Copper. Denn ist dieses Bit gelöscht, wartet der Copper bei jedem Wait, bis der Blitter seine Aufgaben beendet hat. Anschließend wird mit den anderen Copperbefehlen fortgefahren. Setzt man jedoch dieses Bit, so wird der Blitterstatus ignoriert und der Copper setzt seine Arbeit ungestört fort.

Ein Beispiel:

```
dc.w $0501,$fffe
```

Es wird auf die vertikale Zeile $0500 gewartet.

Der SKIP-Befehl

Der dritte und letzte Copper-Befehl dient dazu, die tatsächliche Strahlenposition mit der im ersten Befehlswort angegebenen zu vergleichen. Ist diese größer oder gleich, so überspringt der Copper den nächsten Befehl. Somit lassen sich einfache Verzweigungen programmieren. Der Aufbau des Skip-Befehls ist dem des Wait-Befehls sehr ähnlich, nur daß Bit 0 im zweiten Befehlswort immer auf 1 zu setzen ist.

3.1.2. DIE COPPERLISTE

Die oben besprochenen Copperbefehle werden in einer Art kleinem Programm zusammengefaßt, der sogenannten Copperliste, und anschließend dem Copper übergeben.
Der Copper läßt sich nicht nur durch die drei Copperbefehle steuern, sondern auch durch einige wichtige Register beeinflussen.

COPCON ($dff02e)
Ist Bit 0, welches auch das einzige Bit dieses Registers ist, gesetzt, so kann der Copper, wie schon oben erwähnt, auch auf die Register von $dff040 bis $dff07e zugreifen.

COP1LCH ($dff080)
Dieses Register enthält das Highword der Adresse der ersten Copperliste.

COP1LCL ($dff082)
Hier ist das, zu COP1LCH zugehörige Lowword der ersten Copperliste gespeichert.

COP2LCH ($dff084)
Dieses Register enthält das Highword der Adresse der zweiten Copperliste.

COP2LCL ($dff086)
Hier ist das zu COP1LCH zugehörige Lowword der zweiten Copperliste gespeichert.

COPJMP1 ($dff088)
Nach jedem abgearbeiteten Befehl erhöht der Copper einen Zähler um zwei Worte, damit dieser auf den neuen Befehl in der Copperliste zeigt. COPJMP1 und COPJMP2 dienen als Zeiger auf diesen Zähler. Das bedeutet, daß man die Anfangsadresse der Copperlist in diesen Zähler laden muß, um den Copper an einer bestimmten Position beginnen zu lassen.

COPJMP2 ($dff08a)
Genau wie COPJMP1, nur für die zweite Copperliste.

Diese Coppersteuer-Register sind nicht die einzigen wichtigen Register. Denn um ein sichtbares Bild zu erzeugen sind noch einige, teils vom Copper unabhänige Schritte durchzuführen. Sehen wir uns doch zuerst das fertige Programm an. Das folgende Listing erstellt eine Copperliste, die einen eigenen Screen öffnet:

```
;
; Copper-Demo für einen einfachen Screen
;
;
s:
         move.w #$4000,$dff09a   ;Interrupts sperren
                                 ;Betriebssystem ausschalten
         move   #$0020,$dff096   ;Sprites ausschalten
         move.l #copperl,$dff084 ;Copperliste aktivieren
;
loop:    move.l $dff004,d0       ;Warten auf Rasterstrahl
         and.l  #$fff00,d0
         cmp.l  #$00003000,d0
         bne.s  loop
         btst   #6,$bfe001       ;Maustaste gedrückt ?
         bne.s  loop             ;Ja ?
;
ende:    move   #$c000,$dff09a   ;Interrupts erlauben
e:       rts                     ;Programmende
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
         dc.w $00e0,$0005 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
```

Zuerst müssen wir das Betriebssystem abschalten, da es bei unserem Spiel nur stören würde. Dies geschieht mit dem Befehl

```
        move.w #$4000,$dff09a
```

der alle Interrupts sperrt, und somit das Betriebssystem 'lahmlegt'.Dann müssen wir die Sprites ausschalten, um ein eventuelles lästiges Flackern zu verhindern. Dies geschieht mit

```
        move    #$0020,$dff096
```

Nun wird die eigene Copperliste dem Copper mitgeteilt. Wir beschreiben das COP2LCH und das COP2LCL Register gleichzeitig mit einem MOVE-Befehl.

```
        move.l #copperl,$dff084
```

Das war eigentlich schon alles.
Damit das Programm auf dem Bildschirm bleibt, müssen wir eine kleine Schleife anlegen, in der immer gewartet wird, bis der Rasterstrahl eine bestimmte Position erreicht hat. Danach wird die linke Maustaste abgefragt. Wurde sie gedrückt, so wird das Programm beendet, wenn nicht wird wieder in die Schleife verzweigt.

```
loop:   move.l $dff004,d0       ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        btst   #6,$bfe001       ;Maustaste gedrückt ?
        bne.s  loop             ;Ja ?
```

Wird nun die linke Maustaste gedrückt, müssen wir das Programm "sauber" beenden. Das heißt wir müssen sämtliche Interrupts wieder erlauben.

```
ende:   move   #$c000,$dff09a   ;Interrupts erlauben
e:      rts                     ;Programmende
```

Nun kommen wir zum eigentlichen Teil unseres Programms, der Copperliste. Sicherlich werden Sie nicht sofort alle Befehle verstehen und ich möchte jetzt auch noch nicht alles erklären, sondern auf das Kapitel Playfields verweisen, wo alle Befehle für den Bildschirm- und Playfieldaufbau erklärt werden. Dennoch ist es für unsere Copper-Demonstration wichtig diese Befehle in unsere Copperliste aufzunehmen.

Es läßt sich allgemein sagen, daß es sich in dieser Copperliste um eine Auflistung von MOVE-Befehlen handelt, die dem Copper zur Bildschirmdarstellung übermittelt werden. Es werden sowohl die Fenstergröße als auch alle dazugehörigen Parameter eingestellt. Einen Befehl, den Sie sicherlich gleich erkannt haben, ist die Zuweisung der Hintergrundfarbe: dc.w $0180,$0000 wobei $0000 schwarz bedeutet. Verändern Sie diesen Wert zum Beispiel einmal auf $0f00 und betrachten Sie das Ergebnis.

Um dem Copper zu sagen, daß die Copperliste beendet ist und er wieder von vorne beginnen soll, bedient man sich einer einfachen Methode: Man läßt ihn auf eine unmögliche Strahlenposition warten. Dadurch erkennt er das "Ende" und beginnt von neuem.

```
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
         dc.w $00e0,$0005 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
;
         dc.w $0180,$0000 ;COLOR00
;
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
```

Wollen wir zum besseren Verständnis der Copperliste und ihrer Anwendung noch ein kleines Programm schreiben, welches uns ermöglicht, die Hintergrundfarbe öfter zu initialisieren. Dadurch können wir den, jetzt einfarbigen Bildschirm in einen mehrfarbigen umwandeln.

Das nun folgende Programm soll die Hintergrundfarbe (Register $dff180) dreimal unterteilen und mit drei neuen Farbwerten belegen.

Bevor wir jedoch die eigene Copperliste starten, werden wir noch eine kleine Veränderung im Hauptprogramm vornehmen. Es kann bei unserem ersten Listing vorgekommen sein, daß auf dem schwarzen Bildschirm zusätzlich irgendein "Mist" zu sehen war. Dies liegt daran, daß wir die Adresse, an der unser Bildschirmspeicher liegt, das ist der Bereich, der von uns als sichtbarer Bereich definiert wurde, nicht von restlichen Daten befreit (gelöscht) haben. Unser Bildschirmspeicher zeigt auf die Adresse $50000
(Befehl: dc.w $00e0,$0005 und dc.w $00e2,$0000 - nähere Erklärungen dazu entnehmen Sie bitte dem Punkt 4.2. Playfields). Um dies zu vermeiden und um ein Überschreiben eventuell, an dieser Stelle, befindlichen Daten zu verhindern, definieren wir unseren Bildschirmspeicher am Ende des Programms. Mit dem Befehl "blk.b" resevieren wir einen 10800 Byte großen Platz, an dem wir später unser Bild ablegen werden. Um nun die Adresse des Bildschirmspeichers in die Copperliste einzutragen, bedienen wir uns folgender Befehle:

```
move.l #bild,d0
move   d0,pl1+6
swap   d0
move   d0,pl1+2
```

Zuerst wird die Adresse des Bildschirmspeichers in d0 eingelesen. Anschließend wird das Lowword in das Register $dff0e2 (= pl1+6) eingetragen. Daraufhin vertauschen wir die Words in d0 (swap d0) und tragen das Highword ein.

Danach widmen wir uns dem Farbwechsel. Alle Copperbefehle sind genau diesselben, wie aus unserem vorhergehenden Beispiel. Der erste Befehl, der die Farbe in das Hintergrundregister schreibt (dc.w $0180,$0f00), wurde von schwarz (aus dem vorhergehenden Beipiel) auf Rot ausgebessert. Anschließend folgt ein Wait-Befehl und eine neuerliche Farbzuweisung, welche die Hintergrundfarbe auf weiß ändert. Es folgt ein weiterer Wait-Befehl und noch eine Farbzuweisung, welche die Farbe wieder auf Rot setzt.

Dieses Beispiel zeigt, was alles mit dem Copper möglich ist, da man jede Rasterzeile einzeln ansprechen kann. Experimentieren Sie mit dem fertigen Listing, um etwas Übung mit dem Wait- und Move-Befehlen zu bekommen.

```
;
;        Copper-Demo
;
;
s:       move.l #bild,d0
         move   d0,pl1+6
         swap   d0
         move   d0,pl1+2
         move.w #$4000,$dff09a   ;Interrupts sperren;
                                 ;Betriebssystem ausschalten
         move   #$0020,$dff096   ;Sprites ausschalten
         move.l #copperl,$dff084 ;Copperliste aktivieren
;
loop:    move.l $dff004,d0       ;Warten auf Rasterstrahl
         and.l  #$fff00,d0
         cmp.l  #$00003000,d0
         bne.s  loop
         btst   #6,$bfe001       ;Maustaste gedrückt ?
         bne.s  loop             ;Ja ?
;
ende:    move   #$c000,$dff09a   ;Interrupts erlauben
e:       rts                     ;Programmende
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
pl1:     dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
;
         dc.w $0180,$0f00 ;COLOR00 -> Rot
         dc.w $7001,$fffe ;WAIT
         dc.w $0180,$0fff ;COLOR00 -> Weiß
         dc.w $e001,$fffe ;WAIT
         dc.w $0180,$0f00 ;COLOR00 -> Rot
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
bild:    blk.b 10800,0
```

3.1.3. FARBSCROLLING MITTELS COPPER

Nun wollen wir den Copper dazu bewegen, die Hintergrundfarbe nicht nur drei- oder viermal, sondern gleich hundertmal oder noch öfter zu ändern und zwar in jeder Zeile. Es gibt mehrere Möglichkeiten, dies zu erreichen: Die eine wäre, wie im vorhergehenden Beispiel beschrieben, jeden Wait-Befehl per Hand in die Copperliste einzutragen. Ich glaube, ich brauche nicht zu erwähnen, wie lange das dauern würde. Eine weitaus schnellere, kürzere und angenehmere Methode ist es, dem Computer die Aufgabe der Copperlisten-Erstellung zukommen zu lassen. Wir definieren lediglich den Platz, wo die Copperliste abgelegt werden soll, sowie die Anzahl und Art der Befehle und lassen den Amiga die Liste berechnen.

Das Listing ist fast identisch mit den bereits oben gezeigten Beispiel-Programmen, nur haben wir die Erstellung der Copperliste in eine eigene Unterroutine gepackt, die mittels

```
        bsr.s  initcopper
```

angesprungen wird. Anschließend wird der Buffer, den wir unter dem Namen "cins" in der Copperliste eingetragen haben, sowie der Buffer mit den Farbwerten in a0 bzw. a1 eingelesen.

```
        lea    cins(pc),a0
        lea    color(pc),a1
```

Jetzt wird die Anzahl der Zeilen in d0 übergeben. Da wir in unserem Fall 100 Zeilen erstellen wollen, müssen wir 99 in d0 schreiben. 99 deshalb, weil wir weiter unten im Listing mit dem Befehl "dbf" arbeiten und dieser nicht bei 0 sondern bei -1 verzweigt.

```
        move   #99,d0
```

Außerdem wird die Startzeile festgelegt, indem wir diese in d1 schreiben.

```
        move   #$6001,d1
```

Nun können wir mit unserer Schleife beginnen und nacheinander die Befehle, die wir in unserer Copperliste haben wollen, eintragen.

Zuerst wird das erste Befehlswort des Wait-Befehls (in d1) in den Copperlisten-puffer eingetragen ((a0)+) und anschließend das zweite.

Dann wird das erste Befehlswort des Move-Kommandos des Farbregisters (#$0180) definiert. Bevor wir nun die Farbe in den Buffer schreiben, überprüfen wir, ob der Buffer mit den Farbwerten schon fertig ausgelesen wurde. Wir fragen ganz einfach eine bestimmte Endekennung ab. In unserem Fall $ffff. Sind die Farben schon fertig ausgelesen, werden sie wieder neu an den Anfang der Liste gesetzt. Wenn nicht, wird die nächste Farbe in die Copperliste eingetragen.

```
coppercopy:
        move    d1,(a0)+
        move    #$fffe,(a0)+
        move    #$0180,(a0)+
        cmp     #$ffff,(a1)
        bne.s   daher
        lea     color(pc),a1
daher:  move    (a1)+,(a0)+
```

Ehe die Schleife abgeschlossen werden kann, muß natürlich der Wait-Befehl erhöht werden, damit der Copper die nächste Farbe auch wirklich in der nächsten Zeile initialisiert. Dies geschieht durch einfaches Addieren von

```
        add     #$0100,d1
```

in d1. Nun können wir mit dem Befehl "dbf" die Schleife und mit "rts" den Rücksprung aus der Subroutine beenden.

```
        dbf     d0,coppercopy
        rts
```

Jetzt ist noch der, oben schon erwähnte, Buffer für die Copperliste anzulegen. Die Größe läßt sich ganz einfach berechnen: Da wir vier Befehlswörter und 100 Zeilen verwenden, muß unser Buffer 400 words groß sein.

Abschließend finden Sie das komplette Listing, welches 100 Zeilen der Hintergrundfarbe, mittels Copper, mit jeweils einer Farbe aus einem eigenen Farbbuffer beschreibt.

```
;
;   Copper-Farben-Demo
;
;
s:
        move.w #$4000,$dff09a       ;Interrupts sperren;
                                    ;Betriebssystem ausschal-
                                     ten
        move   #$0020,$dff096       ;Sprites ausschalten
        move.l #copperl,$dff084     ;Copperliste aktivieren
        move.l #bild,d0
        move   d0,pl1+6             ;Low word
        swap   d0
        move   d0,pl1+2             ;High word
        bsr.s  initcopper           ;Copperliste erstellen
;
loop:   move.l $dff004,d0           ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        btst   #6,$bfe001           ;Maustaste gedrückt ?
        bne.s  loop                 ;Ja ?
;
ende:   move   #$c000,$dff09a       ;Interrupts erlauben
e:      rts                         ;Programmende
;
initcopper:
        lea    cins(pc),a0          ;Copperliste
        lea    color(pc),a1
        move   #99,d0
        move   #$6001,d1
coppercopy:
        move   d1,(a0)+
        move   #$fffe,(a0)+
        move   #$0180,(a0)+
        cmp    #$ffff,(a1)
        bne.s  daher
        lea    color(pc),a1
daher:  move   (a1)+,(a0)+
        add    #$0100,d1
        dbf    d0,coppercopy
        rts
;
```

```
; Farben für Copperliste
color:
      dc.w $000,$111,$222,$333,$444,$555,$666,$777,$888,$999
      dc.w $aaa,$bbb,$ccc,$ddd,$eee,$fff,$eee,$ddd,$ccc,$bbb
      dc.w $aaa,$999,$888,$777,$666,$555,$444,$333,$222,$111
      dc.w $ffff

;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
pl1:     dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
cins:    blk.w 400,0      ;Platzhalter
         dc.w $0180,$0000 ;COLOR00
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
;
bild:    blk.b 10800,0    ;Bildschirmspeicher
```

Als nächstes wollen wir etwas "Leben" in unsere Copperliste bringen, indem wir unsere Farben "scrollen" (bewegen) lassen. Wir wollen das Programm so verändern, daß die 100 Zeilen mit dem grauen Farbverlauf von unten nach oben wandern.

Dazu verändern wir das vorhergehende Programm, indem wir aus der Unter-Routine "initcopper" die Zuweisung der Farben entfernen, da diese Aufgabe von einer anderen Unterroutine übernommen wird. Statt den Farbwerten "moven" wir eine Farbe (in unserem Fall schwarz) in die Copperliste.

Im Hauptprogramm, an der Stelle an welcher der Druck der linken Maustaste abgewartet wird, fügen wir eine neue Unterroutine ein, welche die Aufgabe hat, laufend die Farbwerte in der Copperliste zu verändern. Es sollen praktisch die Farben in der unten aufgelisteten Farbtabelle dargestellt und nach jedem Durchlauf um eine Zeile versetzt werden.

Der Anfang ist ähnlich wie bei unserer alten "initcopper" Routine, denn zuerst wird sowohl der Copperlisten Buffer, als auch die zugehörigen Farben eingelesen und mit der Endekennung verglichen. Zum Copperlisten Buffer muß allerdings 6 hinzuaddiert werden, da wir das zweite Befehlswort des ersten Farbbefehls beschreiben wollen; vorher liegen nämlich das erste und zweite Befehlswort des Wait-Kommandos, sowie das erste der Farbzuweisung. Zusammen ergibt das 6 Byte um an die richtige Adresse zu gelangen. Der Farb-Buffer hat jetzt einen Zeiger erhalten (colorptr), der auf das aktuelle Farbword im Buffer zeigt.

```
scrollcopper:
        lea     cins+6(pc),a0
        move.l  colorptr,a1
        cmp     #$ffff,(a1)
        bne.s   nocol
        move.l  #color,colorptr
        move.l  colorptr,a1
```

Anchließend wird in d0 die Anzahl der Zeile minus 1 übergeben.

```
nocol:  move    #99,d0
```

Nun folgt die Schleife, in der die Farben in die Copperliste kopiert werden. Innerhalb dieser Schleife muß natürlich wieder die Endekennung der Farbtabelle abgefragt werden, um die Farben am Ende der Tabelle wieder zurückzusetzten.

```
scroll: move    (a1)+,(a0)
        add.l   #8,a0
        cmp     #$ffff,(a1)
        bne.s   weiter
        lea     color,a1
weiter: dbf     d0,scroll
```

Abgeschloßen wird die Routine durch Erhöhen des Farbzählers um 2.

```
        add.l   #2,colorptr
        rts
```

Dies sind die Änderungen, die für das Copper-Farbscrolling nötig sind. Das gesamte Listing ist unten nocheinmal zu sehen. Experimentieren Sie mit den Routinen und verändern Sie diese nach Belieben, um den Umgang mit Wait- und Move-Befehlen zu üben.

```
;
;  Copper-Farben-Demo "scrolling"
;
;
s:
        move.w #$4000,$dff09a     ;Interrupts sperren;
                                  ;Betriebssystem ausschal-
                                   ten
        move   #$0020,$dff096     ;Sprites ausschalten
        move.l #copperl,$dff084   ;Copperliste aktivieren
        move.l #bild,d0           ;Bildschirmspeicher
                                  ;eintragen
        move   d0,pl1+6           ;Low word
        swap   d0
        move   d0,pl1+2           ;High word
        bsr.s  initcopper         ;Copperliste erstellen
;
loop:   move.l $dff004,d0         ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        bsr.s  scrollcopper       ;Farben "scrollen"
        btst   #6,$bfe001         ;Maustaste gedrückt ?
        bne.s  loop               ;Ja ?
;
ende:   move   #$c000,$dff09a     ;Interrupts erlauben
e:      rts                       ;Programmende
;
initcopper:
        lea    cins(pc),a0        ;Copperliste
        move   #99,d0
        move   #$6001,d1
```

```
coppercopy:
        move    d1,(a0)+
        move    #$fffe,(a0)+
        move    #$0180,(a0)+
        move    #$0000,(a0)+
        add     #$0100,d1
        dbf     d0,coppercopy
        rts
;
scrollcopper:
        lea     cins+6(pc),a0       ;In jeder Zeile werden
        move.l  colorptr,a1         ;die Farben neu in die
        cmp     #$ffff,(a1)         ;Copperliste eingetragen
        bne.s   nocol
        move.l  #color,colorptr
        move.l  colorptr,a1
nocol:  move    #99,d0
scroll: move    (a1)+,(a0)
        add.l   #8,a0
        cmp     #$ffff,(a1)
        bne.s   weiter
        lea     color,a1
weiter: dbf     d0,scroll
        add.l   #2,colorptr
        rts
;
; Farben für Copperliste
colorptr:       dc.l color
color:
     dc.w $000,$111,$222,$333,$444,$555,$666,$777,$888,$999
     dc.w $aaa,$bbb,$ccc,$ddd,$eee,$fff,$eee,$ddd,$ccc,$bbb
     dc.w $aaa,$999,$888,$777,$666,$555,$444,$333,$222,$111
     dc.w $ffff
;
```

```
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
pl1:     dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
cins:    blk.w 400,0      ;Platzhalter
         dc.w $0180,$0000 ;COLOR00
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
;
bild:    blk.b 10800,0    ;Bildschirmspeicher
```

3.1.4. ERWEITERTES FARBSCROLLING

Nun wollen wir uns eine weitere Variante des Farbscrollings ansehen. Da der Copper nicht nur auf vertikale Positionen (Zeilen) warten kann, sondern auch auf horizontale, ergeben sich viele Möglichkeiten tolle Effekte zu produzieren. Wir wollen die Hintergrundfarbe zweimal in einer Zeile ändern, eine Seite hinauf und die andere hinunter wandern lassen. Damit die Trennlinie zwischen beiden Seiten nicht langweilig gerade aussieht, werden wir sie Sinusförmig anlegen und zudem schneller bewegen, als die Farben. - Klingt kompliziert? Nein, im Gegenteil, es ist ein relativ einfacher Effekt mit großer Wirkung. Sehen Sie sich das fertige Listing zuerst an, und probieren Sie es aus, damit Sie das Programm besser verstehen.

```
;
;       Copper-Scroll-Effekt
;
s:
        move.w #$4000,$dff09a       ;Interrupts sperren;
                                    ;Betriebssystem ausschal-
                                     ten
        move   #$0020,$dff096       ;Sprites ausschalten
        move.l #copperl,$dff084     ;Copperliste aktivieren
        move.l #pic,d0              ;Bild-Adresse in d0
        move   d0,plane1+6          ;Low word eintragen
        swap   d0
        move   d0,plane1+2          ;High word eintragen
        bsr.s  initcopper           ;Copperliste initialisie
                                     ren
;
loop:   move.l $dff004,d0           ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        bsr.s  scrollcopper         ;Copper "scrollen"
        btst   #6,$bfe001           ;Maustaste gedrückt ?
        bne.s  loop                 ;Ja ?
;
ende:   move   #$c000,$dff09a       ;Interrupts erlauben
e:      rts                         ;Programmende
;
```

```
initcopper:
        lea     cins(pc),a0         ;Copperliste erstellen
        move    #255,d0
        move    #$0001,d1
        move    #$0001,d2
cloop:  move    d1,(a0)+
        move    #$fffe,(a0)+
        move.l  #$01800000,(a0)+
        move    d2,(a0)+
        move    #$fffe,(a0)+
        move    #$0180,(a0)+
        move    #$0000,(a0)+
        add     #$0100,d1
        add     #$0100,d2
        dbra    d0,cloop
        rts
;
scrollcopper:
        lea     cins+6(pc),a0       ;Farben "scrollen"
        lea     cins+4078(pc),a2
nomincol:
        move.l  waitptr,a1
        cmp.b   #$00,(a1)
        bne.s   noend
        move.l  #wait,waitptr
        move.l  waitptr,a1
noend:  move.l  colorptr,a3
        cmp.w   #$ffff,(a3)
        bne.s   noendr
        move.l  #color,colorptr
        move.l  colorptr,a3
noendr: move    #254,d0
doit:   move    (a3),(a0)
        move.b  (a1)+,d5
        sub.b   #10,d5
        or.b    #$01,d5
        move.b  d5,3(a0)
        add.l   #$10,a0
        move    (a3),(a2)
        sub.l   #$10,a2
        cmp.b   #$00,(a1)
        bne.s   noend1
        lea     wait(pc),a1
noend1: add.l   #2,a3
        cmp     #$ffff,(a3)
        bne.s   noendr1
        lea     color(pc),a3
```

```
noendr1:
        dbra    d0,doit
        add.l   #2,waitptr
        add.l   #2,colorptr
        rts
;
waitptr:        dc.l wait
wait:   dc.b 150,144,137,132,126,120,115,110,105,100
        dc.b 96,93,89,86,84,82,81,80,80,80,81,82,84,86
        dc.b 89,93,96,100,105,110,115,120,126,132,137
        dc.b 144,150,156,162,168,174,179,185,190,195
        dc.b 199,204,207,211,213,216,218,219,219,220
        dc.b 220,219,218,216,213,211,207,204,199,195
        dc.b 190,185,179,174,168,162,156,0,0
;
colorptr:       dc.l color
color:
dc.w $f0f,$f1f,$f2f,$f3f,$f4f,$f5f,$f6f,$f8f,$f9f,$faf,$fbf
dc.w $fcf,$fdf,$fef,$fff,$fff,$fef,$fdf,$fcf,$fbf,$faf,$f9f
dc.w $f8f,$f6f,$f5f,$f4f,$f3f,$f2f,$f1f,$f0f,$ffff
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
cins:    blk.w 2048,0     ;Copper-Buffer
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
;
pic: blk.b 10800,0   ;Buffer für Bildschirmspeicher
;
```

Nun werden Sie feststellen, daß sich eigentlich nicht sehr viel zu den vorhergehenden Listings geändert hat. Nur die "scrollcopper" und "initcopper" Routinen sind etwas länger geworden. Bei "initcopper" wurden noch jeweils ein Wait- und ein Farb-Move-Befehl eingefügt.

Um eine Farbe nach oben zu scrollen, haben wir einfach die Farbwerte in den eingelesenen Copperbuffer geschrieben. Damit sich die Farben in die andere Richtung bewegen, müssen wir lediglich die Farbwerte von hinten in die Copperliste eintragen. Dies geschieht durch:

```
scrollcopper:
        lea    cins+6(pc),a0       ;Farben "scrollen"
        lea    cins+4078(pc),a2
```

Nun folgen die schon bekannten Abfragen der Endekennung und die Übergabe der zu bewegenden Zeilen in d0.

```
nomincol:
        move.l waitptr,a1
        cmp.b  #$00,(a1)
        bne.s  noend
        move.l #wait,waitptr
        move.l waitptr,a1
noend:  move.l colorptr,a3
        cmp.w  #$ffff,(a3)
        bne.s  noendr
        move.l #color,colorptr
        move.l colorptr,a3
noendr: move   #254,d0
```

Jetzt wird die Schleife definiert, in der sowohl die Farbwerte als auch die Wait-Befehle in die Copperliste eingetragen werden.

```
doit:   move   (a3),(a0)
        move.b (a1)+,d5
        sub.b  #10,d5
        or.b   #$01,d5
        move.b d5,3(a0)
        add.l  #$10,a0
        move   (a3),(a2)
        sub.l  #$10,a2
```

Wie vorher, folgen nun die Überprüfungen der Endekennungen und abschließend das Addieren zu den Farb- und Wait-Zeigern.

```
        cmp.b   #$00,(a1)
        bne.s   noend1
        lea     wait(pc),a1
noend1: add.l   #2,a3
        cmp     #$ffff,(a3)
        bne.s   noendr1
        lea     color(pc),a3
noendr1:
        dbra    d0,doit
        add.l   #2,waitptr
        add.l   #2,colorptr
        rts
```

Das war das ganze Geheimnis. Sie sehen, daß man mit ganz einfachen und kurzen Programmen die erstaunlichsten Effekte erzielen kann.

3.1.5. Die 3D-Rolle

Als letzten Copper-Effekt, der auch in unserem Spiel Anwendung finden soll, wollen wir uns die Programmierung einer 3D-Rolle ansehen. Diese Rolle wird in dem Spiel den Hintergrund, quasi die hinterste Ebene bilden. Das hat überhaupt keinen Einfluß auf die Handlung des Spieles, sondern dient der optischen Gestaltung.

Zur Aufgabenstellung: Es soll eine 30 Zeilen hohe, optisch dreidimensional wirkende, "Rolle" gebildet werden, die den Eindruck erwecken soll, als ob sie rotiere. Zudem soll sie auf dem Bildschirm von oben nach unten wandern.

Beginnen wir wieder mit der Erstellung der Copperliste. An der Grundstruktur dieser Liste ändert sich nicht viel. Wir benötigen legiglich 30 Zeilen, in jeder Zeile einen Wait- und einen Move-Befehl.

```
initcopper:     lea     cins(pc),a0
                move.l #29,d0
                move.w #$c001,d1
copycop:        move.w d1,(a0)+
                move.w #$fffe,(a0)+
                move.w #$0180,(a0)+
                move.w #$0000,(a0)+
                add.w   #$0100,d1
                dbra    d0,copycop
                rts
```

Darüberhinaus müssen wir den Buffer für den benötigten Speicher in die Copperliste eintragen. Dieser ist bei 30 Zeilen zu je 4 word breiten Befehlen 120 words groß.

Nun kommen wir zum schwierigeren Teil, nämlich der "scrollcopper"-Routine, die nicht nur die Rolle dreht, sondern auch den 3D-Effekt erzeugt und letztendlich auch noch bewegt.
Sehen Sie sich doch zuerst das fertige Listing an, die Erklärungen folgen danach.

```
scrollcopper: lea     cins(pc),a0
              move.l  parptr,a4
              cmp.w   #$ffff,(a4)
              bne.s   tres
              move.l  #parabel,parptr
              move.l  parptr(pc),a4
tres:         move.l  shadeptr(pc),a3
              cmp.w   #$ffff,(a3)
              bne.s   qwex
              move.l  #shade,shadeptr
              move.l  shadeptr(pc),a3
qwex:         move.l  colorptr(pc),a2
              cmp.w   #$ffff,(a2)
              bne.s   ewewx
              move.l  #colors,colorptr
              move.l  colorptr(pc),a2
ewewx:        move    #29,d0
              move.w  (a4),d5
iix:          move    d5,(a0)
              add     #$0100,d5
              add.l   #6,a0
              move    (a2)+,d7
              move    (a3)+,d6
              and.w   d6,d7
              move    d7,(a0)
              cmp.w   #$ffff,(a2)
              bne.s   werwerx
              lea     colors(pc),a2
werwerx:      cmp.w   #$ffff,(a3)
              bne.s   trex
              lea     shade(pc),a3
trex:         add.l   #2,a0
              dbra    d0,iix
              add.l   #2,colorptr
              add.l   #2,parptr
              rts
```

Im Wesentlichen ähnelt dieses Listing ebenfalls den schon besprochenen. Wie man einen 30 Zeilen hohen Farbverlauf darstellt und diesen zum "Scrollen" bringt, wissen Sie jetzt schon. Aber wie müssen wir nun vorgehen, wenn wir den gewünschten 3D-Effekt erzielen wollen.
Beginnen wir ganz von vorne: Wann sieht man quasi 3D?
Dieser Effekt entsteht, wenn man die Ecken (in unserem Fall die oberen und unteren Zeilen unserer Rolle) etwas abdunkelt. Dadurch erhält unsere Rolle einen plastischen Eindruck. Aber wie programmiert man dies?

Ganz einfach, man legt neben der Tabelle für die gewünschten Farben, eine zweite Tabelle, mit einer Abstufung von Grauwerten, an. Da die Rolle an den Rändern (gemeint sind die oberen und unteren Zeilen) dunkler werden soll, müssen die Grauwerte an diesen Stellen eben immer dunkler werden.
Sehen wir uns zunächst die benötigten Farbtabellen an.

```
;
colorptr: dc.l colors
colors: dc.w $0f00,$0f00,$0f00,$0f00,$0f00
        dc.w $0f00,$0f00,$0f00,$0f00,$0f00
        dc.w $0fff,$0fff,$0fff,$0fff,$0fff
        dc.w $0fff,$0fff,$0fff,$0fff,$0fff
        dc.w $0f00,$0f00,$0f00,$0f00,$0f00
        dc.w $0f00,$0f00,$0f00,$0f00,$0f00
        dc.w $ffff
;
shadeptr: dc.l shade
shade:
      dc.w $111,$222,$333,$444,$555,$666,$777,$888,$999,$aaa
      dc.w $bbb,$ccc,$ddd,$eee,$fff,$fff,$eee,$ddd,$ccc,$bbb
      dc.w $aaa,$999,$888,$777,$666,$555,$444,$333,$222,$111
      dc.w $ffff
```

In der Tabelle "colors" sind die eigentlichen Farben für die Rolle, in diesem Fall Rot-Weiß-Rot, enthalten. In der zweiten Tabelle hingegen befinden sich die Farben, die von dunklem Grau, über helles Weiß bis wieder dunkles Grau reichen. Um nun den 3D-Effekt zu erzielen, verknüpfen wir diese beiden Farbinformationen miteinander und schreiben das Ergebnis in die Copperliste.

```
move   (a2)+,d7
move   (a3)+,d6
and.w  d6,d7
move   d7,(a0)
```

In a2 befindet sich die eigentliche Farbe der Rolle, in a3 der Verlauf für die Verdunkelung. Wir entnehmen diesen beiden Registern je ein word und verknüpfen beide miteinander. Abschließend wird das Ergebnis in die in a0 stehende Copperliste eingetragen.

Jetzt wollen wir unsere fertige Rolle noch auf und ab bewegen. Dieses Problem lösen wir ganz einfach, indem wir die Wait-Befehle verändern. Damit das Auf- und Abfallen nicht gleichmäßig geschieht, legen wir eine Tabelle mit unterschiedlichen Waits ab. Befindet sich die Rolle oben am Monitor, so soll sie länger stehen bleiben, sich nach unten zu aber schneller bewegen. Dadurch entsteht der Eindruck, sie würde fallen. Sehen Sie dazu folgende Tabelle an:

```
parptr: dc.l parabel
parabel:
dc.w $3001,$3001,$3001,$3001,$3201,$3201,$3201,$3301,$3301
dc.w $3301,$3501,$3501,$3501,$3601,$3601,$3601,$3701,$3701
dc.w $3701,$3801,$3801,$3801,$3901,$3901,$3901,$3a01,$3a01
dc.w $3a01,$3b01,$3b01,$3b01,$3c01,$3c01,$3c01,$3d01,$3d01
dc.w $3d01,$3e01,$3e01,$3e01,$3f01,$3f01,$3f01,$4001,$4001
dc.w $4201,$4201,$4401,$4401,$4601,$4601,$4801,$4801,$4b01
dc.w $4b01,$4e01,$4e01,$5101,$5101,$5401,$5401,$6001,$6001
dc.w $7001,$7001,$8001,$8001,$9001,$9001,$a001,$a001,$b001
dc.w $b001,$c001,$c001,$d001,$d001,$e001,$e001
dc.w $e001,$e001,$d001,$d001,$c001,$c001,$b001,$b001,$a001
dc.w $a001,$9001,$9001,$8001,$8001,$7001,$7001,$6001,$6001
dc.w $5401,$5401,$5101,$5101,$4e01,$4e01,$4b01,$4b01,$4801
dc.w $4801,$4601,$4601,$4401,$4401,$4201,$4201,$4001,$4001
dc.w $3f01,$3f01,$3f01,$3e01,$3e01,$3e01,$3d01,$3d01,$3d01
dc.w $3c01,$3c01,$3c01,$3b01,$3b01,$3b01,$3a01,$3a01,$3a01
dc.w $3901,$3901,$3901,$3801,$3801,$3801,$3701,$3701,$3701
dc.w $3601,$3601,$3601,$3501,$3501,$3501,$3301,$3301,$3301
dc.w $3201,$3201,$3201,$3001,$3001,$3001,$3001
dc.w $ffff
```

Mit diesem kleinen Effekt wollen wir das Kapitel Copper beenden. Es sollte Ihnen zeigen, welche Effekte allein nur mit dem Copper möglich sind. Wie für alles andere auch, gilt besonders hier, viel zu experimentieren, um die nötige Praxis zu erlangen.

3.2. DIE PLAYFIELDS

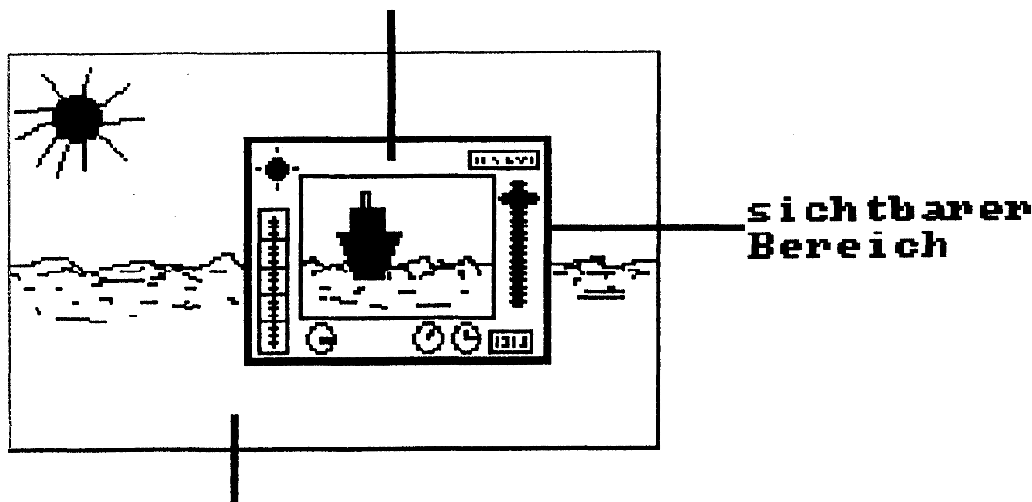
Auf den vorhergehenden Seiten haben wir viel über die Arbeit mit dem Copper gehört. Nun wollen wir uns mit der Bildschirmdarstellung befassen. Es gibt mehrere Möglichkeiten, Objekte bzw. Ebenen auf dem Bildschirm zu zeigen:

- o maximal 6 Planes auf bis zu zwei von einander unabhänigen Playfields, mit max. 4096 Farben.
- o 8 maximal 16 Punkte (Pixel) breite und beliebig hohe Objekte (Sprites), mit maximal drei Farben.
- (o beliebig große Objekte (Bobs) mit beliebig vielen Farben.)

Zu beachten ist dabei, daß die sogenannten Bobs eigenlich keine eigenen Objekte sind, da sie nur mit Hilfe des Blitters, daher auch der Name Blitter Objects (Bobs), in das jeweilige Playfield hineinkopiert werden und somit ein Bestandteil desselben sind. Darum wurde dieser Punkt auch in Klammer gesetzt.

Der Amiga stellt uns für einen Bildschirm (screen) maximal sechs Bitplanes zur Verfügung, mit denen wir unseren Screen gestalten können. Ebenso gibt es verschiedene Auflösungen, in denen diese Bitplanes, jedoch nur eingeschränkt, verwendet werden können. Dazu kommen wir etwas später noch ausführlicher. Außerden können diese Bitplanes, wie schon erwähnt, zu zwei, voneinander unabhänigen, Playfields kombiniert werden. Im Dual-Playfield-Modus kann ein Playfield maximal drei Bitplanes besitzen. Playfield 1 erhält die ungeraden, Playfield 2 die geraden Planes. Wird eine ungerade Anzahl von Bitplanes verwendet, so erhält Playfield 1 um eine Plane mehr. Durch diesen Modus ergeben sich, vor allem für Spiele, sehr viele Möglichkeiten, da diese Playfields wie zwei eigene Screens verwendet werden können. So ergibt sich die Möglichkeit, etwa im Vordergrund eine Anzeigentafel zu gestalten, während im Hintergrund die Landschaft bewegt wird (siehe Bild unten). Zudem kann man ein oder beide Playfields überdimensionieren, das bedeutet, daß nur ein kleiner Teil im sichtbaren Bereich liegt und der Rest hineinbewegt (gescrollt) werden muß.

1. Playfield: z.B.: Anzeigetafel



2. Playfield: z.B.: Hintergrund

Bild 3: Der Dual-Playfield Modus

3.2.1. DIE PLAYFIELD-REGISTER

Um überhaupt ein Playfield auf dem Bildschirm darstellen zu können, müssen einige Register initialisiert werden. Dazu liefert uns der Amiga die folgenden Speicherstellen:

DIWSTRT ($dff08e)

In diesem Register wird die Startposition des Fensters eingestellt. Der Beginn des Bildschirmfensters ist leider auf das linke obere Bildschirm-Viertel beschränkt, da die MSBs der vertikalen und horizontalen Anfangsposition als null angenommen werden.

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

Wie sie aus dem Schema ersehen können, bestimmen die Bits H0-H7 die horizontale Startposition und die Bits V0-V7 die vertikale. Für unsere Copperliste verwenden wir eine Startposition von vertikal 48 und horizontal 129, hexadezimal umgerechnet ergibt das, einen Wert von $3081.

DIWSTOP ($dff090)

Im Gegensatz, zur gerade erklärten Funktion, wird in dieses Register die Endposition eingetragen.

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

Ein weiterer Unterschied ist, daß in diesem Register Bit H8 als 1 angenommen wird, wobei die Endposition in dem Bereich von 256 bis 458 liegen kann. Um eine vertikale Endposition sowohl größer als auch kleiner als 256 zu erhalten, hat man zu einem kleinen Trick gegriffen. Um das V8 Bit auf null zu setzten, schreibt man eine 1 in V7, will man V8 auf 1, so löscht man V7. Damit sind Positionen von 128 bis 312 möglich. Hat man nun den sichtbaren Bereich mittels dieser beiden Register eingestellt, so wird außerhalb, sozusagen als Rahmen die Hintergrundfarbe dargestellt.

DDFSTRT ($dff092)

Hat man dem Amiga mitgeteilt, wie groß das entsprechende Fenster sein soll, so muß man ihn darauf hinweisen, woher er die Daten aus dem Speicher holen soll.
Die Display-Data-Start und Display-Window-Start Register sind voneinander abhängig. Da jede Bitplane im Lores-Modus alle 8 Zyklen eingelesen wird (im Hires-Modus sind es nur 4 Zyklen), und die Hardware noch einen halben Buszyklus zum darstellen der Daten benötigt, besteht eine Differenz von 8.5 Zyklen. Daher errechnet sich der richtige DDFSTRT folgendermaßen.

DDFSTRT: $81/2 - 8.5 = $38

Statt $81 ist der entsprechende Wert aus DIWSTRT einzusetzen.

DDFSTOP ($dff094)

Um den Display-Data-Fetch-Stop Wert zu berechnen, nimmt man einfach die horizontale Auflösung, dividiert diese durch 2, subtrahiert 8 und addiert den DDFSTRT Wert.

DDFSTOP: $38 + (320/2 - 8) = $d0

$38 entspricht dem oben errechneten DDFSTRT Wert und 320 ist die Auflösung (Pixel pro Zeile).

BPL1PTH - BPL6PTL ($dff0e0 - $dff0f8)

Wir haben dem Amiga bereits mitgeteilt, wie groß unser sichtbares Fenster ist, jetzt fehlt nur noch die Adresse, an der die tatsächlichen Bitplane-Informationen im Speicher liegen. Dazu stehen uns 12 Register zur Verfügung, wobei immer zwei die Adresse für eine Bitplane enthalten. Nämlich in Low- und Highword getrennt. Je nachdem, wieviele Planes verwendet werden sollen, müssen die BPLxPTH / BPLxPTL Pointer verwendet werden.

BPLCON0 ($dff100)

Dieses Register hat eine wichtige Funktion bei der Bildschirmdarstellung, denn es legt die Auflösungen fest und bestimmt, wieviele Bitplanes verwendet werden. Sehen wir uns die einzelnen Bits etwas genauer an:

Bit	Name	Funktion
15	HIRES	Ist dieses Bit auf 1 gesetzt, so wird der hochauflösende Modus (640 Punkte pro Zeile) eingeschalten.
14	BPU2	Die Bits von BPU2, BPU1 und BPU0 ergeben zusammen eine 3-Bit Zahl, die angibt, wieviele Planes eingeschalten sind. In hexadezimaler Schreibweise ist dies leicht zu erkennen, da an der ersten Stelle einfach eine Zahl von 1 bis 6 eingetragen werden muß. Z.B.: $5000 entspricht 5 aktivierten Planes.
13	BPU1	siehe BPU2
12	BPU0	siehe BPU2
11	HOMOD	Will man den Hold-and-Modify Modus aktivieren, so ist dieses Bit auf 1 zu setzten. In diesem Modus ist es möglich alle 4096 Farben des Amiga gleichzeitig darzustellen.
10	DBPLF	Ist dieses Bit auf 1, so ist der Dual-Playfield Modus aktiviert, und es werden die geraden und ungeraden Planes wie zwei unterschiedliche Bildschirme behandelt. Die Bits 10 und 11 dürfen nicht gemeinsam aktiv sein. Will man jedoch den Extra-Half-Bright Modus (64 Farben) aktivieren, so müssen die Bits 10 und 11 auf 0 gesetzt und alle 6 Planes eingeschalten sein.
9	COLOR	Dieses Bit schaltet den Videofarbausgang von Agnus ein.
8	GAUD	Mit diesem Bit wird das Genlock Audio eingeschalten.

7	-	unbenützt
6	-	unbenützt
5	-	unbenützt
4	-	unbenützt
3	LPEN	Will man einen Lightpen abfragen, so ist dieses Bit auf 1 zu setzen.
2	LACE	Um den Interlace Modus zu benützen, muß dieses Bit gesetzt werden.
1	ERSY	Hat dieses Bit den Wert 1, werden die Anschlüße für horizontale und vertikale Synchronisation von Ausgang auf Eingang umgeschalten. Dadurch kann das Amigabild durch externe Signale synchronisiert und zu jedem beliebigen Fernsehbild gemischt werden. Dieses Bit wird vor allem von Genlock-Interfaces benützt.
0	-	unbenützt

BLPCON1 ($dff102)

Dieses Register enthält die Stellungen der geraden und ungeraden Planes, die mittels diesem Register um 15 Pixel positioniert werden können. Um die Arbeitsweise des Registers zu verstehen, ist hier die Bitbelegung aufgelistet:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	--	--	--	--	--	--	--	--	P2	P2	P2	P2	P1	P1	P1	P1

Die Bits 8-15 sind frei. Die verbleibenden 8 Bits teilen sich die geraden und ungeraden Planes auf, wobei P1 die geraden und P2 die ungeraden Planes symbolisieren. Dieses Register wird auch zum Scrolling verwendet, aber dazu kommen wir etwas später.

BPLCON2 ($dff104)

In diesem Register werden die Prioritäten der Sprites und Playfields bestimmt. Im Kapitel Sprites werden die Prioritäten noch ausführlich besprochen.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	--	--	--	--	--	--	--	--	--	PP	SP	SP	SP	SP	SP	SP

Die Bits 7 bis 15 sind nicht belegt. PP, Bit 6 ist verantwortlich für die Priorität der geraden zu den ungeraden Planes. Es bestimmt, welche im Vorder- und welche im Hintergrund liegen. Natürlich findet dieses Bit nur im Dual-Playfield Modus Verwendung.

BPL1MOD ($dff108)

Dieses Register bestimmt den Modulo Wert der ungeraden Planes. Was ist nun der Modulo Wert? Mit diesem Wert ist es möglich, sogenannte rechteckige Speicherbereiche zu definieren. Auf dieses Prinzip werden Sie beim Amiga noch öfter stoßen, z.B.: beim Blitter. So ist es möglich, innerhalb eines großen Speicherabschnittes, einen kleineren zu bestimmen, der eine eigene Höhe und Breite besitzt. Ein Beispiel verschafft Klarheit: Wir haben unser Bildschirmfenster mit einer Breite von 320 Pixel, das sind 40 Bytes (20 Worte), definiert. Das daraufliegende Playfield ist aber 640 Pixel breit (= 80 Byte = 40 Worte). In den BPLxPT Registern stehen nun die Startadressen unserer Planes. Da nach dem Einlesen einer Zeile dieser Wert vom Computer automatisch um 20 Worte erhöht wird, würde er an der falschen Adresse die nächste Zeile einlesen, denn richtigerweise müßte er 40 Worte hinzuaddieren. Steht nun ein Wert in den Modulo Registern, wird dieser automatisch hinzuaddiert, damit die nächste Zeile richtig eingelesen wird. Logischerweise errechnet sich der Modulo Wert einer Plane aus der Differenz der beiden unterschiedlichen Zeilenlängen.

BPL2MOD ($dff10a)

Dies ist das zweite Modulo Register, das aber nur für die geraden Planes gültig ist.

COLOR00 - COLOR31 ($dff180 - $dff1be)

In diesen Registern werden den einzelnen Planes die Farben zugewiesen. Wie schon in vorhergehenden Kapiteln erwähnt, verwendet der Amiga die sogenannte 'Additive' Farbmischung. Es werden die drei Grundfarben Rot, Grün und Blau gemischt. Dadurch entstehen alle anderen Farben. Beim Amiga gibt es pro Farbe 16 Helligkeitsstufen. 16 hoch 3 ergibt 4096 verschiedene Farbtöne. Nun wird, je nach verwendeter Anzahl von Bitplanes und den darauf gesetzten oder gelöschten Bits, eine Farbe gebildet, und dem Farbregister zugewiesen.

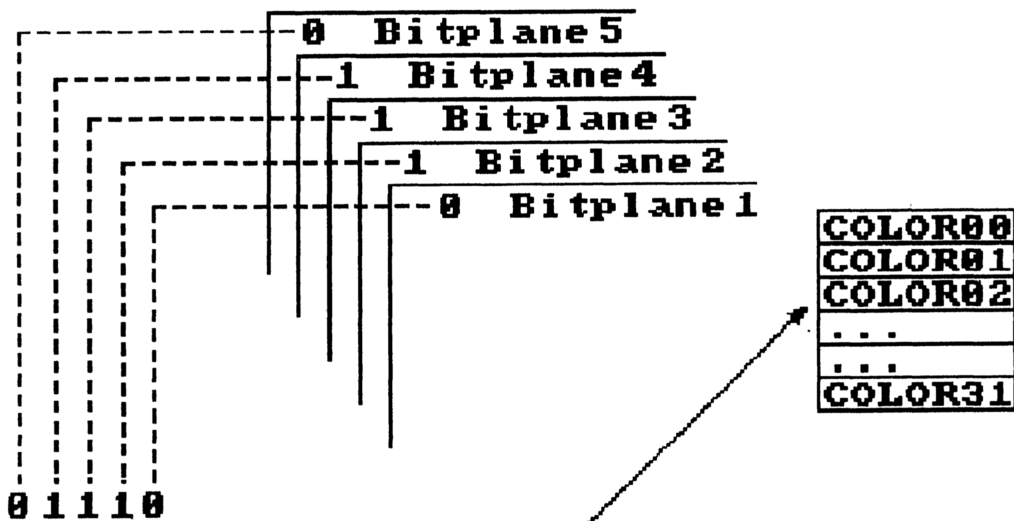


Bild 4: Die Farbdarstellung

Wie schon erwähnt, gibt es 32 solcher Farbregister, die je einen 12 Bit Farbwert aufnehmen können. Das erste Farbregister COLOR00 dient als Hintergrund und, falls das Fenster kleiner ist, als Rahmenfarbe. Sehen wir uns kurz die Bit-Belegung eines Farbregisters an:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	--	--	--	--	R	R	R	R	G	G	G	G	B	B	B	B

Die Bits 12 bis 15 sind nicht belegt. R, G und B entspricht einem jeweiligen Rot-, Grün- und Blau-Wert. Zum besseren Verständnis sehen wir uns einmal die hexadezimale Schreibweise an, da diese etwas übersichtlicher ist.



Bild 5: RGB-Werte

Wurde der Extra-Halfbrite-Modus aktiviert, stehen uns 64 Farben zur Verfügung, obwohl wir nur 32 Farbregister haben. Ermöglicht wird dies durch den Gebrauch der sechsten Bitplane. Wir erinnern uns: Bei 5 Bitplanes haben wir 32 Farben (2 hoch 5 = 32). Nimmt man nun die sechste Bitplane hinzu, so erhält ma 2 hoch 6 = 64 Werte. Da aber nur 32 Farbregister zur Verfügung stehen, wird wieder mit einem Trick gearbeitet. Soll nun eine Farbe der sechste Plane dargestellt werden, so wird diese mit der halben Helligkeit gezeigt. Dadurch benötigt man nur 32 Farbregister, kann aber dennoch 64 Farben verwenden.

3.2.2. DIE DARSTELLUNG VON PLAYFILEDS

Nachdem wir die Register, die zur Bildschirmdarstellung wichtig sind, kennengelernt haben, befassen wir uns als nächstes mit der Representation auf dem Monitor. Es soll ein Bild mit nur einer Plane (=2 Farben) gezeigt werden. Dazu müssen wir dieses zuerst mit Hilfe eines Malprogramms, z.B. DPaint III zeichnen. Da Grafiken meistens als IFF-File abgespeichert werden, müssen wir diese in ein, für den Video-Chip, lesbares Format bringen. Das IFF-Format ist ein Standard der sich besonders auf dem Amiga durchgesetzt hat, da es Bilder komprimiert abspeichert. Um diese aber für unsere Zwecke lesbar zu machen, müssen wir sie wieder zurück, in das sogenannte RAW-Format, konvertieren. Möglich ist dies mit dem im Verlag Gabriele Lechner erschienenen CONVERT-UTILITY, das diese Arbeit übernimmt. Ein weiteres aber leider etwas teures Programm ist das Malprogramm Pixmate. Wollen Sie kein eigenes Bild erstellen, so können Sie die auf der Beispiel-Diskette mitgelieferten Bilder verwenden.

Wie wird aber ein, im Speicher liegendes, Bild auf dem Bildschirm dargestellt? Verwendet man nur eine Plane, wie in unserem Fall 1, so entspricht jedes gesetzte Bit einem Bildschirmpunkt. Es läßt sich jetzt auch ganz leicht errechnen, wieviel Platz das Bild im Speicher benötigt. Da unsere Grafik 320 Punkte breit ist, entspricht dies 320 Bit. 320 Bit sind 40 Byte, da ein Byte bekanntlich 8 Bit besitzt (320/8=40). Weil wir 256 Zeile anzeigen, ergibt dies einen Speicherbedarf von 40*256=10240 Byte. Diesen Bereich müssen wir mittels einer blk.b Anweisung in unserem Programm definieren. Als nächstes sehen wir uns die Copperliste an, die für die Darstellung verantwortlich ist. Die ersten acht Words, dienen zur Einstellung der Bildschirmgröße.

```
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
```

Nun folgt die Angabe der Bitplanes und des Modus mit dem Register $dff100:

```
         dc.w $0100,$1200 ;BPLCON0
```

Die weiteren Register sind alle auf null zu setzten.

```
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
```

Die nachfolgenden BPLxPT zeiger in der Copperliste zwar auf null,

```
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
```

aber wie Sie vielleicht aus dem Listing ersehen, wird vor dem Start der richtige Wert in die Copperliste eingetragen. Das machen wir deshalb, weil unser Programm nicht immer an der gleichen Adresse liegt, und sich daher natürlich auch die Adresse des Bildbuffers ändert. Wäre die angegebene Adresse fix, so würde das Bild nicht korrekt angezeigt werden. Daher wird die neue Adresse wie folgt in die Copperliste eingetragen:

```
         move.l #pic,d0          ;Bild-Adresse in d0
         move   d0,plane1+6      ;Low word eintragen
         swap   d0
         move   d0,plane1+2      ;High word eintragen
```

Zuerst wird das Low-Word in das Register $dff0e2 und anschließend das High-Word in $dff0e0 geschrieben. Abschließend bestimmen wir die Farben, die unsere Plane haben soll. In $dff180 wird die Hintergrundfarbe ($000 = Schwarz) eingetragen, die an allen Stellen, an denen die Bits unserer Plane gelöscht sind, sichtbar wird. An den anderen Positionen, an denen Bits gesetzt sind, erscheint die Farbe des Registers $dff182 ($fff = Weiß). Unser Listing ist somit fertig.

Wenn Sie das Listing ausprobieren, müssen Sie in das fertig assemblierte Programm das Bild von der Diskette in den Speicher, und zwar genau an die mit "pic" definierte Adresse laden, damit es überhaupt sichtbar wird. Beim Seka-Assembler übernimmt dies das RI (Read Image) Kommando, mit dem fehlende Module an bestimmte Adressen geladen werden können.

```
SEKA>ri (RETURN)
FILENAME>df0:picture (RETURN)
BEGIN>pic (RETURN)
END> (RETURN)
```

Haben Sie das obenstehende Kommando ausgeführt, wird das Bild von der Diskette nachgeladen und in unserem Bildschirmspeicher abgelegt. Nun kann das Programm mit dem "j" Kommando angesprungen und das fertige Bild betrachtet werden.

```
;
; Darstellung eines Bildes 320*256; 1Plane = 2Farben
;
;
s:
         move.w #$4000,$dff09a    ;Interrupts sperren;
                                  ;Betriebssystem ausschalten
         move   #$0020,$dff096    ;Sprites ausschalten
         move.l #copperl,$dff084  ;Copperliste aktivieren
         move.l #pic,d0           ;Bild-Adresse in d0
         move   d0,plane1+6       ;Low word eintragen
         swap   d0
         move   d0,plane1+2       ;High word eintragen
;
loop:    move.l $dff004,d0        ;Warten auf Rasterstrahl
         and.l  #$fff00,d0
         cmp.l  #$00003000,d0
         bne.s  loop
         btst   #6,$bfe001        ;Maustaste gedrückt ?
         bne.s  loop              ;Ja ?
;
ende:    move   #$c000,$dff09a    ;Interrupts erlauben
e:       rts                      ;Programmende
;
```

```
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0100,$1200 ;BPLCON0
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
;
pic:     blk.b 10240      ;Buffer für Bild
;
```

Wie funktioniert die Darstellung eines Bildes mit vielen Farben? Dazu benötigen wir mehrere Planes. Wir haben festgestellt, daß jedes gesetzte Bit einer Farbe entspricht. Allerdings benötigen wir nicht für jede Farbe eine neue Bitplane, sondern die Anzahl der zu verwendenden Farben steigt exponentiell. Wie schon oben erwähnt gibt es maximal 6 Bitplanes. Das heißt, maximal 2 hoch 6 = 64 Farben, Ausnahme ist der Hold-and-Modify Modus, kurz HAM genannt. Bei fünf Bitplanes, wie unser nächstes Beispiel zeigt, gibt es 32 verschiedene Farben, und wir verwenden alle 32 Farbregister.

Wie aus dem Listing zu erkennen ist, wurde erstens der Bildschirmspeicher (pic) vergrößert. Da wir 5 Planes verwenden, benötigen wir auch fünfmal soviel Speicher (10240*5=51200). Außerdem wurde ein Label mit der Größe einer Plane definiert. Wie wir bereits wissen, liegen die 5 Planes hintereinander im Speicher. Beim Eintrag in die Copperliste wird zum Datenregister d0 die Größe (10240) hinzuaddiert, um die Adresse der nächsten Plane zu erhalten.

Die Copperliste selbst wurde natürlich um die restlichen Farb- und BPLxPT-Register erweitert. Darüberhinaus wurde die Anzahl im BPLCON0-Register auf 5 erhöht.

```
         dc.w $0100,$5200 ;BPLCON0
```

Sonst ist alles gleich geblieben. Assemblieren Sie das Programm und lesen Sie das Bild an der vorgesehenen Adresse ein.

```
;
; Darstellung eines Bildes 320*256; 5Planes = 32Farben
;
;
s:
        move.w #$4000,$dff09a   ;Interrupts sperren;
                                ;Betriebssystem ausschalten
        move   #$0020,$dff096   ;Sprites ausschalten
        move.l #copperl,$dff084 ;Copperliste aktivieren
        move.l #pic,d0          ;Bild-Adresse in d0
        move   d0,plane1+6      ;Low word eintragen
        swap   d0
        move   d0,plane1+2      ;High word eintragen
        swap   d0
        add.l  #size,d0         ;Größe addieren
        move   d0,plane2+6      ;Low word eintragen
        swap   d0
        move   d0,plane2+2      ;High word eintragen
        swap   d0
        add.l  #size,d0         ;Größe addieren
        move   d0,plane3+6      ;Low word eintragen
        swap   d0
        move   d0,plane3+2      ;High word eintragen
        swap   d0
        add.l  #size,d0         ;Größe addieren
        move   d0,plane4+6      ;Low word eintragen
        swap   d0
        move   d0,plane4+2      ;High word eintragen
        swap   d0
        add.l  #size,d0         ;Größe addieren
        move   d0,plane5+6      ;Low word eintragen
        swap   d0
        move   d0,plane5+2      ;High word eintragen
;
loop:   move.l $dff004,d0       ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        btst   #6,$bfe001       ;Maustaste gedrückt ?
        bne.s  loop             ;Ja ?
```

```
;
ende:     move   #$c000,$dff09a    ;Interrupts erlauben
e:        rts                      ;Programmende
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$5200 ;BPLCON0
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
plane2:  dc.w $00e4,$0000 ;BPL2PTH
         dc.w $00e6,$0000 ;BPL2PTL
plane3:  dc.w $00e8,$0000 ;BPL3PTH
         dc.w $00ea,$0000 ;BPL3PTL
plane4:  dc.w $00ec,$0000 ;BPL4PTH
         dc.w $00ee,$0000 ;BPL4PTL
plane5:  dc.w $00f0,$0000 ;BPL5PTH
         dc.w $00f2,$0000 ;BPL5PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $0184,$08c3 ;COLOR02
         dc.w $0186,$0a67 ;COLOR03
         dc.w $0188,$0bcb ;COLOR04
         dc.w $018a,$0875 ;COLOR05
         dc.w $018c,$054c ;COLOR06
         dc.w $018e,$0b5a ;COLOR07
         dc.w $0190,$0865 ;COLOR08
         dc.w $0192,$0645 ;COLOR09
         dc.w $0194,$00ff ;COLOR10
         dc.w $0196,$0ff0 ;COLOR11
         dc.w $0198,$0f0f ;COLOR12
         dc.w $019a,$000f ;COLOR13
         dc.w $019c,$00f0 ;COLOR14
         dc.w $019e,$0f00 ;COLOR15
         dc.w $01a0,$0000 ;COLOR16
         dc.w $01a2,$0111 ;COLOR17
         dc.w $01a4,$0222 ;COLOR18
         dc.w $01a6,$0333 ;COLOR19
```

```
          dc.w $01a8,$0444 ;COLOR20
          dc.w $01aa,$0555 ;COLOR21
          dc.w $01ac,$0666 ;COLOR22
          dc.w $01ae,$0777 ;COLOR23
          dc.w $01b0,$0888 ;COLOR24
          dc.w $01b2,$0999 ;COLOR25
          dc.w $01b4,$0aaa ;COLOR26
          dc.w $01b6,$0bbb ;COLOR27
          dc.w $01b8,$0ccc ;COLOR28
          dc.w $01ba,$0ddd ;COLOR29
          dc.w $01bc,$0eee ;COLOR30
          dc.w $01be,$0fff ;COLOR31
          dc.w $ffff,$fffe ;Warten auf unmögliche
                           ;Position
;
size = 320/8*256
pic:      blk.b 51200      ;Buffer für Bild
;
```

3.2.3. DER "SCHUNKEL"-EFFEKT

Sehen wir uns das Register BLPCON1 ($dff102) noch einmal etwas näher an. Wie schon in der Register-Beschreibung erwähnt wurde, dient es zum horizontalen Verschieben der Bitplanes. Die Bits 0-3 sind für die geraden, die Bits 4-7 für die ungeraden Planes zuständig. Wir können dieses Register für einen interessanten Effekt verwenden: Die Bildschirmabbildung soll hin- und her-"geschunkelt" werden. Wir wissen inzwischen, daß eine Bewegung bis zu 15 Pixel nach rechts erzielt werden kann. Jetzt soll das dargestellte Bild von unten nach oben, mit einer sinus-förmigen Wellenlinie, durchlaufen werden. Dadurch entsteht der Eindruck, daß das Bild durchgeschüttelt (geschunkelt) wird.

Wie funktioniert dies? Zuerst müssen wir eine, wie vom Farbscrolling her bekannte, Copperliste erstellen, nur wird nach jedem Wait-Befehl, der auf die nächste Zeile wartet, nicht ein Farbregister eingetragen, sondern das BLPCON1-Register. Damit wären die Vorbereitungen für den Schunkel-Effekt bereits getroffen. Wir haben jetzt eine Copperliste erstellt, in welcher in jeder Zeile ein neuer Move-Befehl steht. Wir können dadurch in jeder Zeile von neuem bestimmen, um wieviele Punkte unser Bildschirm verschoben werden soll. Dies geschieht in der Unterroutine "scrollcopper".

Hier wird ein Sinus-Wert aus der Tabelle "sinus" entnommen und in BLPCON1 eingetragen. Damit es so aussieht, als ob die Sinus-Wellen nach oben wandern, müssen wir den Zeiger auf diese Tabelle (sinusptr), nach jedem Durchlauf, um 2 erhöhen.

Die vielen Null-Byte am Anfang der Tabelle sind dafür gedacht, daß das Bild nicht ununterbrochen "geschunkelt" wird, sondern einige Zeit stillsteht, und erst anschließend mit dem Effekt wieder begonnen wird (denn 0 bedeutet keine Verschiebung!).

```
;
; Darstellung eines Bildes 320*256; 5Planes = 32Farben
; mit "Schunkel-Effekt"
;
s:
      move.w #$4000,$dff09a    ;Interrupts sperren;
                               ;Betriebssystem ausschalten
      move   #$0020,$dff096    ;Sprites ausschalten
      move.l #copperl,$dff084  ;Copperliste aktivieren
      move.l #pic,d0           ;Bild-Adresse in d0
      move   d0,plane1+6       ;Low word eintragen
      swap   d0
      move   d0,plane1+2       ;High word eintragen
      swap   d0
      add.l  #size,d0          ;Größe addieren
      move   d0,plane2+6       ;Low word eintragen
      swap   d0
      move   d0,plane2+2       ;High word eintragen
      swap   d0
      add.l  #size,d0          ;Größe addieren
      move   d0,plane3+6       ;Low word eintragen
      swap   d0
      move   d0,plane3+2       ;High word eintragen
      swap   d0
      add.l  #size,d0          ;Größe addieren
      move   d0,plane4+6       ;Low word eintragen
      swap   d0
      move   d0,plane4+2       ;High word eintragen
      swap   d0
      add.l  #size,d0          ;Größe addieren
      move   d0,plane5+6       ;Low word eintragen
      swap   d0
      move   d0,plane5+2       ;High word eintragen
      bsr.s  initcopper        ;Copperliste initialisieren
;
loop: move.l $dff004,d0        ;Warten auf Rasterstrahl
      and.l  #$fff00,d0
      cmp.l  #$00003000,d0
      bne.s  loop
      bsr.s  scrollcopper      ;Bild "schunkeln"
      btst   #6,$bfe001        ;Maustaste gedrückt ?
      bne.s  loop              ;Ja ?
;
ende: move   #$c000,$dff09a    ;Interrupts erlauben
```

```
e:     rts                         ;Programmende
;
initcopper:
       lea     cins(pc),a0         ;Copperliste erstellen
       move.w #355,d0
       move.w #$00e1,d1
coploop:
       move.w d1,(a0)+
       move.w #$fffe,(a0)+
       move.w #$0102,(a0)+
       move.w #$0000,(a0)+
       add.w   #$0100,d1
       dbra    d0,coploop
       rts
;
scrollcopper:
       lea     cins+6(pc),a0       ;Bild "schunkeln"
       move.l sinusptr,a1          ;Die Sinus-Daten werden
       cmp.b   #$ff,(a1)           ;aus der Tabelle eingelesen
       bne.s   nosinus             ;und in jeder Zeile in
       move.l #sinus,sinusptr      ;BLPCON1 ($dff102) Register
       move.l sinusptr,a1          ;geschrieben.
nosinus:
       move.w #355,d0
sinn:  moveq   #0,d3
       move.w (a1)+,d3
       move.w d3,d4
       lsl.b   #4,d3
       add.w   d4,d3
       move.w d3,(a0)+
       add.l   #6,a0
       cmp.b   #$ff,(a1)
       bne.s   nosin
       lea     sinus(pc),a1
nosin:
       dbra    d0,sinn
       add.l   #$2,sinusptr
       rts
;
sinusptr: dc.l sinus               ;Sinus-Daten
```

```
sinus:
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 dc.b 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
;
 dc.b 1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
```

```
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b 1,1,1,1,2,2,3,3,4,4,5,5,6,6
 dc.b 7,8,8,9,9,10,10,11,11,12,12,13,13,13,13,14,14,14,14
 dc.b 14,14,14,13,13,13,13,12,12,11,11,10,10,9,9,8,8
 dc.b 7,6,6,5,5,4,4,3,3,2,2,1,1,1,1,0,0,0,0,0,0,0
 dc.b $FF,0
;
copperl:
         dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0064 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$5200 ;BPLCON0
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
plane2:  dc.w $00e4,$0000 ;BPL2PTH
         dc.w $00e6,$0000 ;BPL2PTL
plane3:  dc.w $00e8,$0000 ;BPL3PTH
         dc.w $00ea,$0000 ;BPL3PTL
plane4:  dc.w $00ec,$0000 ;BPL4PTH
         dc.w $00ee,$0000 ;BPL4PTL
plane5:  dc.w $00f0,$0000 ;BPL5PTH
         dc.w $00f2,$0000 ;BPL5PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $0184,$08c3 ;COLOR02
         dc.w $0186,$0a67 ;COLOR03
         dc.w $0188,$0bcb ;COLOR04
         dc.w $018a,$0875 ;COLOR05
         dc.w $018c,$054c ;COLOR06
```

```
          dc.w $018e,$0b5a ;COLOR07
          dc.w $0190,$0865 ;COLOR08
          dc.w $0192,$0645 ;COLOR09
          dc.w $0194,$00ff ;COLOR10
          dc.w $0196,$0ff0 ;COLOR11
          dc.w $0198,$0f0f ;COLOR12
          dc.w $019a,$000f ;COLOR13
          dc.w $019c,$00f0 ;COLOR14
          dc.w $019e,$0f00 ;COLOR15
          dc.w $01a0,$0000 ;COLOR16
          dc.w $01a2,$0111 ;COLOR17
          dc.w $01a4,$0222 ;COLOR18
          dc.w $01a6,$0333 ;COLOR19
          dc.w $01a8,$0444 ;COLOR20
          dc.w $01aa,$0555 ;COLOR21
          dc.w $01ac,$0666 ;COLOR22
          dc.w $01ae,$0777 ;COLOR23
          dc.w $01b0,$0888 ;COLOR24
          dc.w $01b2,$0999 ;COLOR25
          dc.w $01b4,$0aaa ;COLOR26
          dc.w $01b6,$0bbb ;COLOR27
          dc.w $01b8,$0ccc ;COLOR28
          dc.w $01ba,$0ddd ;COLOR29
          dc.w $01bc,$0eee ;COLOR30
          dc.w $01be,$0fff ;COLOR31
cins:     blk.w 1424,0     ;"Schunkel"-Buffer
          dc.w $ffff,$fffe ;Warten auf unmögliche
                           ;Position
;
size = 320/8*256
pic: blk.b 51200,0         ;Buffer für Bild
;
```

3.2.4. SCROLLING

Vertikales Scrolling

Nachdem wir uns die grundlegenden Informationen über den Aufbau der Playfields und deren Verwendung zueigen gemacht haben, kommen wir nun zu einem, bei weitem interessanteren Kapitel, dem Bewegen derselben in alle Richtungen. Der Fachausdruck dafür ist 'Scrolling'.

Wir unterscheiden zwei verschiedene Bewegungsrichtungen - die horizontale und die vertikale. Beginnen wir mit der einfacheren von beiden, die in unserem Spiel vorkommt, der vertikalen Richtung.

Zuerst müssen wir uns einig sein, welches Format die zu scrollende Bitplane, bzw. Bitplanes haben sollen. Nehmen wir als Beispiel gleich eine Plane aus unserem Spiel an. Da wir dabei zwei verschiedene Hintergrund Playfields (im Dusal-Playfield-Modus) verwenden, die in unterschiedlichen Geschwindigkeiten bewegt werden sollen, nehmen wir zur besseren Demonstration die kleinere von beiden. Sie besteht aus 3 Bitplanes (8 Farben) und hat die Größe 320 mal 400. Wenn Sie ein eigenes Playfield malen wollen verwenden Sie bitte ein beliebiges Malprogramm, z.B. DPAINT III, und ändern Sie die Fenstergröße und die Anzahl der Planes (Farben). Anschließend können Sie Ihrer Phantasie bei der Gestaltung der Grafiken freien Lauf lassen. Haben Sie diesen Schritt beendet, speichern Sie Ihr Bild auf einer Diskette ab und konvertieren es mit einem Konvertierprogramm, z.B. CONVERT-UTILITY aus dem Verlag Gabriele Lechner, vom IFF in das RAW Format. Danach sollte Ihr Playfield eine Größe von 16000 Bytes auf der Diskette belegen (Größe einer Plane 320 durch 8 mal 400 Zeilen = 16000). Wollen Sie sich diesen aufwendigen Schritt vorerst ersparen, so können Sie die auf der Programmdiskette abgespeicherten Grafiken verwenden.

Haben wir ein fertiges Bild, können wir zur eigentlichen Arbeit übergehen. Wie wir bereits wissen, wird nicht das Playfield selbst verschoben, sondern der Bildschirmausschnitt. Dieser wird von uns in den Zeigern auf die Planes festgelegt (BPLxPT). Das bedeutet wir müssen bei vertikalem Scrolling lediglich diese Zeiger um eine Zeile erhöhen und schon wandert unsere Plane über den Schirm.

Da eine Zeile 320 Punkte breit ist müssen wir die BPLxPT um 40 Byte erhöhen, damit wir in die nächste Zeile gelangen. Wollen wir ein Scrolling in die Gegenrichtung bewirken, so müssen wir diesen Wert subtrahieren.

Ein kleines Problem stellt sich uns noch: Da das Low- und High-Word der BPLxPT-Pointer getrennt in die Register $dff0e0 und $dff0e2 (nur für Plane 1) geschrieben werden, müssen wir aufpassen, daß der Übertrag in unserem Scrolling ebenfalls beachtet wird. Nehmen wir folgende Werte an:

```
dc.w $00e0,$0005
dc.w $00e2,$ff00
```

Wird nun solange in $00e2 addiert, bis dieses Register den Wert $60000 übersteigt, müssen wir die entsprechende Zahl auch in $00e0 korrigieren. Um eine ständige Überprüfung zu vermeiden, lesen wir die Zahl zuerst aus den Registern

```
                lea     pl1+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #40,d0
```

und bilden in einem Datenregister (in unserem Fall d0) ein Long-Word mit der richtigen Adresse der Bitplane und addieren/subtrahieren mit einem Long-Word-Befehl. Somit wird jeder Übertrag automatisch korrekt ausgeführt. Das Ergebnis der Rechnung wird wieder in High- und Low-Word zerlegt und in die passenden Register der Copperliste eingetragen.

```
scrolling_kl:   tst.b   kennbyte1     ;Scrolling des kleinen
                beq.s   rauf          ;Playfields
                lea     pl1+2(pc),a1  ;(Hintergrund)
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #40,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
```

```
        lea     pl2+2(pc),a1
        move    (a1),d0
        swap    d0
        move    4(a1),d0
        add.l   #40,d0
        move    d0,4(a1)
        swap    d0
        move    d0,(a1)

        lea     pl3+2(pc),a1
        move    (a1),d0
        swap    d0
        move    4(a1),d0
        add.l   #40,d0
        move    d0,4(a1)
        swap    d0
        move    d0,(a1)
        add     #1,count1
        cmp     #145,count1
        bne.s   noscroll
        clr.b   kennbyte1
rauf:   add     #1,count1
        lea     pl1+2(pc),a1
        move    (a1),d0
        swap    d0
        move    4(a1),d0
        sub.l   #40,d0
        move    d0,4(a1)
        swap    d0
        move    d0,(a1)

        lea     pl2+2(pc),a1
        move    (a1),d0
        swap    d0
        move    4(a1),d0
        sub.l   #40,d0
        move    d0,4(a1)
        swap    d0
        move    d0,(a1)
```

```
                lea     pl3+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #40,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
                cmp     #290,count1
                bne.s   noscroll
                clr     count1
                move.b  #$01,kennbyte1
noscroll:       rts
;
```

Um die beiden Richtungen unterscheiden zu können, haben wir zwei Labels definiert, in denen vermerkt wird, in welche Richtung die Planes gerade bewegt werden. Dazu dient "kennbyte1". Ist darin eine 1 enthalten, so wird das Playfield hinunter-, ist eine 0 enthalten so wird es hinauf-bewegt. Außerdem wurde der Zähler "count1" definiert, in dem die schon "gescrollten" Zeilen vermerkt werden. Wird eine vorgegebene Anzahl erreicht, so wechselt durch Setzen oder Löschen von "kennbyte1" die Richtung.

Als nächstes wollen wir die, schon erwähnte zweite, größere Ebene unseres Spiels ebenso scrollen, wie die erste, nur etwas schneller. Um das Playfield doppelt so schnell zu bewegen, addieren wir nicht eine sondern zwei Zeilen zu den BPLxPT. Außerdem muß natürlich die Anzahl der Zeilen im "count2" anders definiert werden, da die zweite Plane 600 Zeilen lang ist.

```
scrolling_gr:   tst.b   kennbyte2 ;Scrolling des großen
                beq.s   rauf2  ;Playfields
                lea     pl11+2(pc),a1 ;(Vordergrund)
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #80,d0
                move    d0,4(a1)
```

```
            swap    d0
            move    d0,(a1)
            lea     pl22+2(pc),a1
            move    (a1),d0
            swap    d0
            move    4(a1),d0
            add.l   #80,d0
            move    d0,4(a1)
            swap    d0
            move    d0,(a1)

            lea     pl33+2(pc),a1
            move    (a1),d0
            swap    d0
            move    4(a1),d0
            add.l   #80,d0
            move    d0,4(a1)
            swap    d0
            move    d0,(a1)
            add     #1,count2
            cmp     #173,count2
            bne.s   noscroll2
            clr.b   kennbyte2
rauf2:      add     #1,count2
            lea     pl11+2(pc),a1
            move    (a1),d0
            swap    d0
            move    4(a1),d0
            sub.l   #80,d0
            move    d0,4(a1)
            swap    d0
            move    d0,(a1)

            lea     pl22+2(pc),a1
            move    (a1),d0
            swap    d0
            move    4(a1),d0
            sub.l   #80,d0
            move    d0,4(a1)
            swap    d0
            move    d0,(a1)
```

```
                lea     pl33+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
                cmp     #346,count2
                bne.s   noscroll2
                clr     count2
                move.b  #$01,kennbyte2
noscroll2:      rts
```

Horizontales Scrolling

Um ein horizontales Scrolling zu programmieren, bedarf es sorgfältiger Überlegung. Wenn wir dazu die Startadresse der Bitplane verschieben wollen, so ist dies leider nur um ein Word, das sind 16 Pixel, möglich. Das bedeutet, daß unser Scrolling fürchterlich "ruckeln" würde.

Um dem Abhilfe zu schaffen und unsere Planes sanft und weich zu bewegen, greifen wir auf das, schon bekannte, Register BPLCON1 zurück. Wir wissen bereits, daß mit diesem Register eine Verschiebung von 0 bis 15 Pixel zulässig ist. Damit wäre die Verwendung schon erklärt: Zuerst werden die Planes solange mit Hilfe von BPLCON1 verschoben, bis sie 15 Pixel erreicht haben. Anschließend wird ein Word zur Plane-Adresse addiert und BPLCON1 wieder auf 0 gesetzt.

3.2.5. DIE FERTIGE BILDSCHIRMDARSTELLUNG

Nachdem wir auf den vorangegangenen Seiten die Funktion des Coppers und die Darstellung von Playfields genauer besprochen haben, wenden wir uns nun der Erstellung der fertigen Copperliste und damit der Bildschirmdarstellung unseres Spiels zu.

Unser Spiel soll, wie schon erwähnt, zwei Ebenen haben, auf denen sich der Spieler bewegen kann. Dazu verwenden wir natürlich den Dual-Playfield Modus. Beide Ebenen sollen horizontal scrollen und zwar in verschiedenen Geschwindigkeiten. Die größere Ebene soll Ausmaße von 320 * 600, die kleinere 320 * 400 haben. Der sichtbare Bereich für den gesamten Schirm soll auch in den PAL-Bereich hineinreichen. Das Scrolling selbst wird hingegen nur im NTSC-Bereich stattfinden. Dazu definieren wir erst einmal die Fenstergröße:

```
copperl:
        dc.w $008e,$3081 ;DIWSTRT
        dc.w $0090,$35c1 ;DIWSTOP
        dc.w $0092,$0038 ;DDFSTRT
        dc.w $0094,$00d0 ;DDFSTOP
```

Darüberhinaus können wir die folgenden Register auf 0 setzen, da wir weder eine Verschiebung noch Modulo benötigen.

```
        dc.w $0102,$0000 ;BPLCON1
        dc.w $0108,$0000 ;BPL1MOD
        dc.w $010a,$0000 ;BPL2MOD
```

Im nächsten Move werden die Prioritäten, die Bitplanes betreffend, festgelegt. Deren Bedeutung wird genauer im Kapitel Sprites besprochen, daher ist vorerst folgender Wert anzunehmen:

```
        dc.w $0104,$0064 ;BPLCON2
```

Im nun folgenden Register wird der Dual-Playfield Modus und die Anzahl der verwendeten Bitplanes angegeben. Da unsere beiden Spielflächen aus jeweils drei Bitplanes bestehen, müssen wir alle sechs verfügbaren Bitplanes aktivieren.

```
        dc.w $0100,$6600 ;BPLCON0
```

Die nächsten 12 Einträge sind die Zeiger auf die Adressen, der im Speicher liegenden Bitplanes. Zeiger eins ($00e0) weist auf das High-Word von Bitplane 1 des ersten Playfields, Zeiger zwei auf das Low-Word. Hingegen wird im BPL2PT-Register die Adresse von Bitplane 1 des zweiten Playfields angegeben. Das heißt, es zeigt immer ein Zeiger abwechselnd auf je eine Plane eines Playfields. Vorerst sind die Adressen auf 0 gesetzt. Die richtige Adresse wird jedoch am Anfang unseres Programms eingetragen, dazu wurden auch die Label definiert (pl1,...).

```
pl1:    dc.w $00e0,$0000 ;BPL1PTH
        dc.w $00e2,$0000 ;BPL1PTL
pl11:   dc.w $00e4,$0000 ;BPL2PTH
        dc.w $00e6,$0000 ;BPL2PTL
pl2:    dc.w $00e8,$0000 ;BPL3PTH
        dc.w $00ea,$0000 ;BPL3PTL
pl22:   dc.w $00ec,$0000 ;BPL4PTH
        dc.w $00ee,$0000 ;BPL4PTL
pl3:    dc.w $00f0,$0000 ;BPL5PTH
        dc.w $00f2,$0000 ;BPL5PTL
pl33:   dc.w $00f4,$0000 ;BPL6PTH
        dc.w $00f6,$0000 ;BPL6PTL
```

Als nächstes folgen die Adressen der Sprites. Genaueres über dieses Thema entnehmen Sie bitte dem nächsten Kapitel.

```
p1:     dc.w $0120,$0000 ;SPR0PTH
        dc.w $0122,$0000 ;SPR0PTL
        dc.w $0124,$0000 ;SPR1PTH
        dc.w $0126,$0000 ;SPR1PTL
p2:     dc.w $0128,$0000 ;SPR2PTH
        dc.w $012a,$0000 ;SPR2PTL
        dc.w $012c,$0000 ;SPR3PTH
        dc.w $012e,$0000 ;SPR3PTL
p3:     dc.w $0130,$0000 ;SPR4PTH
        dc.w $0132,$0000 ;SPR4PTL
        dc.w $0134,$0000 ;SPR5PTH
        dc.w $0136,$0000 ;SPR5PTL
p4:     dc.w $0138,$0000 ;SPR6PTH
        dc.w $013a,$0000 ;SPR6PTL
        dc.w $013c,$0000 ;SPR7PTH
        dc.w $013e,$0000 ;SPR7PTL
```

Die nächsten Register sind die schon bekannten Farbregister, in denen die Farben für unsere Playfields, aber auch die Sprites vorhanden sind.

```
       dc.w $0180,$0000   ;Farben für kleines
       dc.w $0182,$00ff   ;Playfield
       dc.w $0184,$00de
       dc.w $0186,$00cc
       dc.w $0188,$00ab
       dc.w $018a,$009a
       dc.w $018c,$0078
       dc.w $018e,$0067
       dc.w $0190,$0000   ;Farben für großes
       dc.w $0192,$0e84   ;Playfield
       dc.w $0194,$0d73
       dc.w $0196,$0b62
       dc.w $0198,$0a51
       dc.w $019a,$0941
       dc.w $019c,$0730
       dc.w $019e,$0620
sprr0:dc.w $01a0,$0000   ;Farben für Joystick-
       dc.w $01a2,$0f0c   ;Sprite
       dc.w $01a4,$0c0a
       dc.w $01a6,$0a08
sprr1:dc.w $01a8,$0000   ;Farben für Feind-
       dc.w $01aa,$0f00   ;Sprite
       dc.w $01ac,$0b00
       dc.w $01ae,$0800
sprr2:dc.w $01b0,$0000   ;Farben für Symbol-
       dc.w $01b2,$00f0   ;Sprite 1
       dc.w $01b4,$00a0
       dc.w $01b6,$0060
sprr3:dc.w $01b8,$0000   ;Farben für Symbol-
       dc.w $01ba,$0ff0   ;Sprite 2
       dc.w $01bc,$0aa0
       dc.w $01be,$0660
```

Um die schon im Kapitel Copper besprochene 3D-Rolle einzubauen, müssen wir dafür einen Platz von 120 Words in der Copperliste definieren. Dies geschieht durch folgenden Befehl:

```
cins: blk.w 120          ;Platz für 3D-Rolle
```

Nun hätten wir den Bildschirmaufbau im oberen Bereich fertiggestellt, da wir aber auch den PAL-Bereich nützen wollen, werden wir dort eine Anzeigetafel plazieren. Dafür werden wir eine 32-farbige (=5 Planes) Anzeigetafel entwerfen und einbauen. Wie können wir aber erneut 5 Planes unterbringen, wenn wir schon alle sechs belegt haben? Ganz einfach, da wir mit den Copper Wait-Befehlen eine beliebige Strahlenposition abfangen können, werden wir warten, bis der Rasterstrahl die letzte Zeile im NTSC-Bereich erreicht hat und anschließend dem Copper erneut ein Playfield zuweisen, das aus 5 Planes besteht und natürlich auch eigene, vom vorhergehenden Playfield unabhänige, Farben besitzen darf. Diese Technik nennt man Bildschirm-Splitting. Dadurch lassen sich praktisch mehrere Ebenen verwenden, obwohl es eigentlich gar nicht soviele gibt.

```
      dc.w $ff01,$fffe    ;Wait
```

Hat der Rasterstrahl die vorgegebene Zeile erreicht, muß dem Copper wieder mitgeteilt werden, wieviele Planes man verwenden möchte. Da wir nur 5 Planes, aber kein Dual-Playfield, verwenden wollen, werden wir folgenden Wert eintragen:

```
      dc.w $0100,$5200    ;BPLCON0
```

Ebenso müssen wir die Adressen der Planes im Speicher festlegen. Noch ist eine Null eingetragen, da die endgültige Adresse zu Beginn des Programms festgelegt wird.

```
pl111:dc.w $00e0,$0000    ;BPL1PTH
      dc.w $00e2,$0000    ;BPL1PTL
      dc.w $00e4,$0000    ;BPL2PTH
      dc.w $00e6,$0000    ;BPL2PTL
      dc.w $00e8,$0000    ;BPL3PTH
      dc.w $00ea,$0000    ;BPL3PTL
      dc.w $00ec,$0000    ;BPL4PTH
      dc.w $00ee,$0000    ;BPL4PTL
      dc.w $00f0,$0000    ;BPL5PTH
      dc.w $00f2,$0000    ;BPL5PTL
```

Jetzt kommen noch die neuen Farben für unser Playfield. Außerdem wurden einige Register durch Labels (col, sieger,...) gekennzeichnet, da wir diese Register vom Programm aus verändern wollen, um Effekte zu erzielen (z.B.: Eine Schrift blinken lassen).

```
col:   dc.w $0180,$0000    ;Farben für die untere
       dc.w $0182,$00f0    ;Anzeigetafel
       dc.w $0184,$0fff
       dc.w $0186,$0fff
sieger:dc.w $0188,$0888
       dc.w $018a,$0256
       dc.w $018c,$0888
       dc.w $018e,$0fff
       dc.w $0190,$0267
       dc.w $0192,$0278
       dc.w $0194,$038a
       dc.w $0196,$03ab
       dc.w $0198,$0d33
       dc.w $019a,$0b33
       dc.w $019c,$0a22
       dc.w $019e,$0922
       dc.w $01a0,$0fff
       dc.w $01a2,$0fff
       dc.w $01a4,$0fff
       dc.w $01a6,$0fff
       dc.w $01a8,$0fff
aa1:   dc.w $01aa,$0f77
aa2:   dc.w $01ac,$0c55
aa3:   dc.w $01ae,$0733
       dc.w $01b0,$0f77
       dc.w $01b2,$0e66
       dc.w $01b4,$0d66
       dc.w $01b6,$0c55
       dc.w $01b8,$0a55
       dc.w $01ba,$0944
       dc.w $01bc,$0844
       dc.w $01be,$0733
```

Abschließend fehlt nur noch die Endekennung, die dem Copper das Ende der Liste signalisiert.

```
       dc.w $ffff,$fffe  ;Endekennung
```

Soweit zum Aufbau unserer Spiel-Copperliste. Weitere Details über Sprites folgen im nächsten Kapitel.

3.3. DIE SPRITES

Wie schon erwähnt, ist der Amiga nicht nur in der Lage sechs Bitplanes, mit maximal zwei Playfields, zu kombinieren, sondern darüberhinaus bis zu acht verschiedene Sprites darzustellen.

Ein Sprite darf eine Breite von maximal 16 Pixel haben. Die Länge ist beliebig und nur durch die Bildschirmgröße begrenzt. Da ein Sprite aus zwei Planes (ähnlich wie bei Playfields) besteht, sind nur drei verschiedene Farben möglich.

Es können auch zwei Sprites miteinander verknüpft werden, um ein 15-farbiges Sprite zu schaffen. Aber darauf gehen wir etwas später noch genauer ein.

3.3.1. AUFBAU UND DARSTELLUNG VON SPRITES

Die Programmierung von Sprites läßt sich auf dem Amiga relativ einfach und komfortabel realisieren. Man erstellt eine Datenliste, welche die Grafik des Objektes enthält, legt diese an eine beliebige Adresse im Speicher und teilt dem Amiga mit, wo er diese Daten findet. Abschließend muß natürlich noch das Sprite-DMA eingeschaltet werden und schon sind unsere Objekte sichtbar.

Der einzige Nachteil ist, daß Sprites eine maximale Breite von 16 Pixel und nur drei Farben haben können. Sehen wir uns zunächst die Sprite-Register an, die zur Darstellung benötigt werden:

SPR0PTH ($dff120)

Dies ist der Zeiger auf das High-Word der im Speicher liegenden Daten von Sprite 0.

SPR0PTL ($dff122)

In diesem Register wird das Low-Word des Zeigers auf die Sprite 0-Daten gespeichert.

SPR1PTx - SPR7PTx ($dff124 - $dff13e)

Für dieses Register gilt dasselbe, wie für die SPR0PTx-Register.

Wir werden diese Zeiger in unserer Copperliste aufnehmen und ähnlich wie bei den Zeigern auf die Playfields, die Zuweisung des richtigen Speichers am Anfang des Programms vornehmen,

```
        move.l #sprite,d0          ;Sprite-Adresse in d0
        move   d0,sp1+6            ;Low word eintragen
        swap   d0
        move   d0,sp1+2            ;High word eintragen
```

indem wir High- und Low-Word getrennt in die Copperliste eintragen. Unter dem Label "sprite" sind die Grafik-Daten definiert, die wir uns als nächstes ansehen wollen: Um ein Sprite zu entwerfen, überlegt man sich am besten zuerst

- o die genaue Größe
- o die Form
- o und die Anzahl der verwendeten Farben

Anschließend kann man beginnen die ungefähre Form des Objektes auf Papier zu bringen. Nehmen wir als Beispiel die Form eines neuen Mauszeigers an. Zeichnen Sie einen Raster, mit maximal 16 Punkten Breite und beliebiger Länge und anschließend die gewünschte Form ein, ohne Rücksicht auf die Farben zu nehmen:

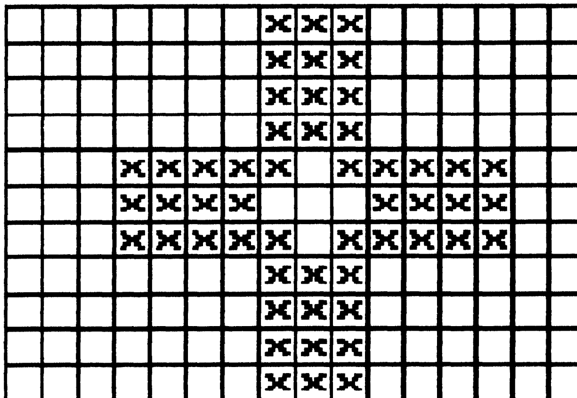


Bild 6: SPRITE-Aufbau

Würden wir dieses Bild nun in binärer Schreibweise als Daten übernehmen, so wäre unser neuer Mauszeiger leider nur einfarbig. Weil wir aber bis zu drei Farben zur Verfügung haben, können wir unsere Grafik colorieren.
Beispiel:

						1	1	3							
						1	2	3							
						1	2	3							
						1	2	3							
		1	1	1	1	1		3	1	1	1	1			
		1	2	2	2				2	2	2	3			
		1	3	3	3	3		3	3	3	3	3			
						1	2	3							
						1	2	3							
						1	2	3							
						1	3	3							

Bild 7: SPRITE-Farben

Doch wie rechnet man diese Grafik in eine, für den Computer verständliche, Form um? Ganz einfach, man kann die in der Grafik eingesetzten Farben wie zwei verschiedene Bitplanes betrachten. Ist ein Bit auf einer Plane gesetzt, so ist auch ein Punkt in der Farbe 1 sichtbar. Will man jedoch Farbe 2, so ist das gewünschte Bit auf der zweiten Plane zu setzen. Ist jedoch Farbe 3 gewünscht, so muß man die Bits beider Planes setzen. Ist weder ein Bit der ersten noch der zweiten Plane gesetzt, so ist die Farbe transparent und alle dahinterliegenden Elemente, ob das nun Playfields oder andere Sprites sind, werden sichtbar. Im Programm schreiben wir die beiden Bitplanes, mit der Dateninformation für unser Sprite, getrennt in binärer Form auf.

```
dc.w %0000001110000000,%0000000010000000
dc.w %0000001010000000,%0000000110000000
dc.w %0000001010000000,%0000000110000000
dc.w %0000001010000000,%0000000110000000
dc.w %0011111011111000,%0000000010000000
dc.w %0010000000001000,%0001110001111000
dc.w %0011111011111000,%0001111011111000
dc.w %0000001010000000,%0000000110000000
dc.w %0000001010000000,%0000000110000000
dc.w %0000001010000000,%0000000110000000
dc.w %0000001110000000,%0000000110000000
```

Viel einfacher ist es jedoch, wenn man das Sprite nicht so umständlich und langwierig auf dem Papier entwirft, sondern dazu ein Malprogramm, wie zum Beispiel DPAINT III verwendet. Nicht nur, daß dieses Programm viel Komfort beim Malen bietet, hat man auch sofort das Ergebnis vor Augen und kann Änderungen leichter und vorallem viel schneller vornehmen als auf dem Papier. Hat man sein Sprite gezeichnet, so kann man es in dieser Form nicht verwenden, sondern es muß ebenfalls in das RAW-Format gebracht werden. Dazu speichern Sie bitte Ihr Sprite als Brush ab und verwenden ein beliebiges Konvertier-Programm. Dadurch sparen Sie nicht nur viel Arbeit, sondern auch viel Zeit.

Eine Einschränkung in der Farbgebung gibt es leider noch. Es stehen nämlich keine eigenen Farb-Register für Sprites zur Verfügung, sondern es werden die Register COLOR16 bis COLOR31 verwendet. Außerdem werden die Sprites paarweise zusammengefaßt. Jeweils zwei Sprites teilen sich vier Farb-Register. Dadurch ergibt sich folgende Aufteilung:

COLOR16	(Farbe durchsichtig)	gilt für Sprite 0 & 1.
COLOR17	(Farbe beliebig)	
COLOR18	(Farbe beliebig)	
COLOR19	(Farbe beliebig)	
COLOR20	(Farbe durchsichtig)	gilt für Sprite 2 & 3.
COLOR21	(Farbe beliebig)	
COLOR22	(Farbe beliebig)	
COLOR23	(Farbe beliebig)	
COLOR24	(Farbe durchsichtig)	gilt für Sprite 4 & 5.
COLOR25	(Farbe beliebig)	
COLOR26	(Farbe beliebig)	
COLOR27	(Farbe beliebig)	
COLOR28	(Farbe durchsichtig)	gilt für Sprite 6 & 7.
COLOR29	(Farbe beliebig)	
COLOR30	(Farbe beliebig)	
COLOR31	(Farbe beliebig)	

Als nächstes sehen wir uns die Positionierung auf dem Bildschirm an. Diese Information wird dem Coputer gleichzeitig mit der Grafik-Datenliste übermittelt. Das erste Long-Word der Datenliste enthält nicht nur die X- und Y-Position des Sprites, sondern auch noch die Länge. Dieses Long-Word wird in zwei Befehlswörter eingeteilt. Das erste Byte des ersten Kontrollwortes enthält die horizontale Startposition (HSTART), das zweite die vertikale Startposition (VSTART). Horizontal ist eine X-Position ein Pixel in Lo-res-Auflösung, vertikal hingegen eine Rasterzeile. Die Positionierung von Sprites ist unabhänig von der Auflösung der Playfields. Die X- und Y-Startposition bezieht sich immer auf die linke obere Ecke unseres Sprites, auch wenn diese als durchsichtig deklariert wurde. Dies ist bei der Programmierung zu berücksichtigen.

Im ersten Byte des zweiten Kontrollwortes wird die vertikale Endposition (VSTOP) festgelegt. Diese entspricht der vertikalen Startposition plus der Anzahl an Zeilen des Sprites. Auf eine horizontale Endposition wurde verzichtet, da jedes Sprite automatisch 16 Punkte breit ist. Falls dies, wie in unserem Beispiel, nicht der Fall ist, so sind einfach die fehlenden Pixel auf null zu setzen.

Will man ein Sprite nicht sichtbar haben, so muß man lediglich die ersten beiden Kontrollworte auf null setzen.

```
dc.w $0000,$0000
```

Dadurch wurde unser Sprite an die Bildschirm-Position X=0, Y=0 gesetzt und ist 0 Zeilen lang. Daher ist es unsichtbar.

Abschließend müssen wir, wie anfangs schon erwähnt, das DMA einschalten, damit unsere Sprites überhaupt dargestellt werden. Dies geschieht mit folgender MOVE-Anweisung:

```
move.w #$8020,$dff096
```

Somit werden alle 8 Sprites aktiviert und, falls sie nicht auf Null gesetzt wurden, auf dem Bildschirm angezeigt. Die genaue Belegung des DMACON-Register entnehmen Sie bitte dem Kapitel Sound-Programmierung.

Nun folgt das fertige Listing, das ein Sprite auf dem Bildschirm darstellt.

```
;
;  Darstellung eines Sprites
;
;
s:
       move.w #$4000,$dff09a    ;Interrupts sperren;
                                ;Betriebssystem ausschalten
       move   #$8020,$dff096    ;Sprites einschalten
       move.l #copperl,$dff084  ;Copperliste aktivieren
       move.l #pic,d0           ;Bild-Adresse in d0
       move   d0,plane1+6       ;Low word eintragen
       swap   d0
       move   d0,plane1+2       ;High word eintragen
       move.l #sprite,d0        ;Sprite-Adresse in d0
       move   d0,sp1+6          ;Low word eintragen
       swap   d0
       move   d0,sp1+2          ;High word eintragen
;
loop:  move.l $dff004,d0        ;Warten auf Rasterstrahl
       and.l  #$fff00,d0
       cmp.l  #$00003000,d0
       bne.s  loop
       btst   #6,$bfe001        ;Maustaste gedrückt ?
       bne.s  loop              ;Ja ?
;
ende:  move   #$c000,$dff09a    ;Interrupts erlauben
e:     rts                      ;Programmende
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $01a2,$0f00 ;COLOR17
         dc.w $01a4,$0a00 ;COLOR18
         dc.w $01a6,$0600 ;COLOR19
```

```
sp1:     dc.w $0120,$0000 ;SPR0PTH
         dc.w $0122,$0000 ;SPR0PTL
         dc.w $0124,$0000 ;SPR1PTH
         dc.w $0126,$0000 ;SPR1PTL
sp2:     dc.w $0128,$0000 ;SPR2PTH
         dc.w $012a,$0000 ;SPR2PTL
         dc.w $012c,$0000 ;SPR3PTH
         dc.w $012e,$0000 ;SPR3PTL
sp3:     dc.w $0130,$0000 ;SPR4PTH
         dc.w $0132,$0000 ;SPR4PTL
         dc.w $0134,$0000 ;SPR5PTH
         dc.w $0136,$0000 ;SPR5PTL
sp4:     dc.w $0138,$0000 ;SPR6PTH
         dc.w $013a,$0000 ;SPR6PTL
         dc.w $013c,$0000 ;SPR7PTH
         dc.w $013e,$0000 ;SPR7PTL
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
;
sprite:  dc.w $a08c,$b000 ;Buffer für Sprite0
         dc.w $0001,$0000
         dc.w $0007,$0001
         dc.w $001A,$0006
         dc.w $0062,$001E
         dc.w $0184,$007C
         dc.w $0604,$01FC
         dc.w $0808,$07F8
         dc.w $0808,$07F8
         dc.w $1010,$0FF0
         dc.w $1010,$0FF0
         dc.w $2060,$1FE0
         dc.w $2180,$1F80
         dc.w $4600,$3E00
         dc.w $5800,$3800
         dc.w $E000,$6000
         dc.w $8000,$0000
;
pic:     blk.b 10240      ;Buffer für Bild
;
```

3.3.2. BEWEGUNG VON SPRITES

Wie wir bereits wissen, wird die Position eines Sprites mit Hilfe der ersten beiden Kontrollworte bestimmt. Will man ein Sprite nun bewegen, so braucht man nur die X- bzw. Y-Position im betreffenden Kontrollwort verändern. Wichtig dabei ist, daß man, wie bei allen anderen Operationen auch, auf das richtige Timing achtet, damit die Sprite-Position nicht verändert wird, wenn der Amiga gerade darauf zugreift. Sehen wir uns das fertige Programm an:

```
;
;  Bewegung eines Sprites
;
;
s:
         move.w #$4000,$dff09a      ;Interrupts sperren;
                                    ;Betriebssystem ausschal-
                                     ten
         move   #$8020,$dff096      ;Sprites einschalten
         move.l #copperl,$dff084    ;Copperliste aktivieren
         move.l #pic,d0             ;Bild-Adresse in d0
         move   d0,plane1+6         ;Low word eintragen
         swap   d0
         move   d0,plane1+2         ;High word eintragen
         move.l #sprite,d0          ;Sprite-Adresse in d0
         move   d0,sp1+6            ;Low word eintragen
         swap   d0
         move   d0,sp1+2            ;High word eintragen
;
loop:    move.l $dff004,d0          ;Warten auf Rasterstrahl
         and.l  #$fff00,d0
         cmp.l  #$00003000,d0
         bne.s  loop
         bsr.s  movesprite          ;Sprite bewegen
         btst   #6,$bfe001          ;Maustaste gedrückt ?
         bne.s  loop                ;Ja ?
;
ende:    move   #$c000,$dff09a      ;Interrupts erlauben
e:       rts                        ;Programmende
;
```

```
movesprite:
         add.b   #1,count           ;Sprite bewegen
         cmp.b   #3,count
         bne.s   nomove
         clr.b   count
         cmp.b   #1,richtung
         beq.s   links
rechts:  add.b   #1,sprite+1
         cmp.b   #$d0,sprite+1
         bne.s   nomove
         move.b  #$1,richtung
nomove:  rts
links:   sub.b   #1,sprite+1
         cmp.b   #70,sprite+1
         bne.s   nomove
         clr.b   richtung
         rts
;
richtung: dc.b 0
count:    dc.b 0
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
planel:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $01a2,$0f00 ;COLOR17
         dc.w $01a4,$0a00 ;COLOR18
         dc.w $01a6,$0600 ;COLOR19
spl:     dc.w $0120,$0000 ;SPR0PTH
         dc.w $0122,$0000 ;SPR0PTL
         dc.w $0124,$0000 ;SPR1PTH
         dc.w $0126,$0000 ;SPR1PTL
sp2:     dc.w $0128,$0000 ;SPR2PTH
         dc.w $012a,$0000 ;SPR2PTL
         dc.w $012c,$0000 ;SPR3PTH
         dc.w $012e,$0000 ;SPR3PTL
```

```
sp3:      dc.w $0130,$0000 ;SPR4PTH
          dc.w $0132,$0000 ;SPR4PTL
          dc.w $0134,$0000 ;SPR5PTH
          dc.w $0136,$0000 ;SPR5PTL
sp4:      dc.w $0138,$0000 ;SPR6PTH
          dc.w $013a,$0000 ;SPR6PTL
          dc.w $013c,$0000 ;SPR7PTH
          dc.w $013e,$0000 ;SPR7PTL
          dc.w $ffff,$fffe ;Warten auf unmögliche
                           ;Position
;
sprite:   dc.w $a08c,$b000 ;Buffer für Sprite0
          dc.w $0001,$0000
          dc.w $0007,$0001
          dc.w $001A,$0006
          dc.w $0062,$001E
          dc.w $0184,$007C
          dc.w $0604,$01FC
          dc.w $0808,$07F8
          dc.w $0808,$07F8
          dc.w $1010,$0FF0
          dc.w $1010,$0FF0
          dc.w $2060,$1FE0
          dc.w $2180,$1F80
          dc.w $4600,$3E00
          dc.w $5800,$3800
          dc.w $E000,$6000
          dc.w $8000,$0000
;
pic:      blk.b 10240      ;Buffer für Bild
;
```

Wichtig ist, daß man die Koordinaten der Sprites in Übereinstimmung mit der Fenstergröße bringt, die in DIWSTRT und DIWSTOP festgelegt wird, da die Sprites nur in diesem Bereich normal dargestellt werden. Außerhalb dieser Begrenzung sind sie nicht mehr sichtbar, da die Daten dort nicht dargestellt werden können.

In dem Label "count" wird die Geschwindigkeit der Bewegung definiert. Vergrößern oder verkleinern Sie diesen Zähler, so werden Sie sehen, mit welchen Geschwindigkeiten sich unser Sprite bewegt. Das Byte "richtung" gibt an, ob sich das Sprite nach links oder nach rechts bewegt, jenachdem ob 0 oder 1 in der Speicherstelle steht.

In unserem Spiel werden wir die Sprites sowohl um einen X- als auch um einen Y-Wert bewegen. Dies bedeutet, daß sie sich mit unterschiedlichen Geschwindigkeiten in verschiedene Richtungen bewegen. Als Beispiel, stellvertretend für alle Bewegungen, wird die Sequenz des roten Punktes ausgewählt.

Zuerst holen wir aus den Speicherplätzen "l5" und "l6" die abgelegten Werte und übergeben sie den Datenregistern d4 und d5.

```
moveenemy:      move.b l5,d4   ;Rotes Sprite bewegen
                move.b l6,d5
```

Anschließend wird der Inhalt von d4 zu "l3" addiert und der horizontale Zähler "l1" um 1 verringert. Ist das Ergebnis ungleich null, so wird zu "label1" verzweigt, wo das Ergebnis in das Kontrollwort eingetragen wird. Wenn Null erreicht ist, wird der Zähler wieder auf 151 gesetzt. Diese Schritte wiederholen sich für die vertikale Position.

```
                add.b  d4,l3
                subq.b #1,l1
                bne.s  label1
                move.b #151,l1
                eor.b  #$fe,d4
label1:         move.b l3,sprite3+1
                add.b  d5,l4
                subq.b #1,l2
                bne.s  label2
                move.b #194,l2
                eor.b  #$fe,d5
label2:         move.b l4,sprite3
                move.b l4,d6
```

Um die VSTOP-Position zu bekommen, wird einfach die Länge des Sprites zur VSTART-Position addiert und in das zweite Kontrollwort eingetragen.

```
add.b  #$10,d6
move.b d6,sprite3+2
```

Abschließend werden die Register d4 und d5 wieder gesichert und die Routine mit rts beendet.

```
move.b d4,15
move.b d5,16
rts
```

Der Amiga bietet uns auch die Möglichkeit einen Sprite-Kanal öfter als einmal zu verwenden, um mehrere Sprites darzustellen. Wie wir bereits wissen, kann ein Sprite beliebig lang sein und ist nur durch die Bildschirmgröße begrenzt. Sobald ein Sprite kürzer ist, können wir ein anderes darunter setzen, indem wir nach der letzten Spritedaten-Zeile wieder zwei neue Kontrollworte einfügen, die die Länge des nächsten Sprites bestimmen.

```
sprite: dc.w $a07c,$b000 ;Buffer für Sprite0
        dc.w $0001,$0000
        dc.w $0007,$0001
        dc.w $001A,$0006
        dc.w $0062,$001E
        dc.w $0184,$007C
        dc.w $0604,$01FC
        dc.w $0808,$07F8
        dc.w $0808,$07F8
        dc.w $1010,$0FF0
        dc.w $1010,$0FF0
        dc.w $2060,$1FE0
        dc.w $2180,$1F80
        dc.w $4600,$3E00
        dc.w $5800,$3800
        dc.w $E000,$6000
        dc.w $8000,$0000
```

```
          dc.w $c09c,$d000 ;Hier liegen die Sprite-
          dc.w $0001,$0000 ;Daten, die ebenfalls
          dc.w $0007,$0001 ;über denselben Kanal
          dc.w $001A,$0006 ;dargestellt werden.
          dc.w $0062,$001E
          dc.w $0184,$007C
          dc.w $0604,$01FC
          dc.w $0808,$07F8
          dc.w $0808,$07F8
          dc.w $1010,$0FF0
          dc.w $1010,$0FF0
          dc.w $2060,$1FE0
          dc.w $2180,$1F80
          dc.w $4600,$3E00
          dc.w $5800,$3800
          dc.w $E000,$6000
          dc.w $8000,$0000
;
```

Wie aus dem Listing ersichtlich ist, wird zuerst das erste Sprite bestimmt und nach der letzten Zeile folgen wieder zwei Kontrollworte, die das nächste Sprite festlegen. So kann man beliebig viele Sprites erzeugen. Diese Sprites können unterschiedliche X-Werte haben, sich jedoch in der vertikalen Richtung nicht überlappen.

Mit einem kleinen Trick ist es möglich den nachfolgenden Sprites jeweils andere Farben zuzuweisen.
Da sich die vertikalen Positionen zwischen den Sprites nicht überschneiden können, braucht man lediglich in der Copperliste einen Wait-Befehl einfügen und in der Zeile, welche die zwei Sprites trennt, neue Farben in die Sprite-Color-Register schreiben. Somit können wir viele verschiedenfarbige Sprites darstellen.

3.3.3. SPRITE-ANIMATIONEN

Wir haben nun gelernt, wie man Sprites darstellt, bewegt, ja sogar wie man mehrere Sprites über einen Kanal verwendet. Als nächstes wollen wir die Sprites für unser Spiel animieren. Das heißt wir werden nicht nur eine Bewegung zeichnen, sondern mehrere und werden diese hintereinander darstellen. Dadurch entsteht der Eindruck, als würde sich unser Objekt beispielsweise verformen.

Wir wissen bereits, daß man, um ein Sprite zu bewegen, einfach die Koordinaten der Kontrollworte verändern muß. Um das Sprite selbst zu verändern, muß man lediglich die Sprite-Grafik-Daten austauschen.

Zuerst definieren wir ein Byte mit dem Namen "animcount", in welchem angegeben wird, wie oft unsere Sprites umgeschalten werden sollen. Es wird alle zehnmal eine andere Animationsphase angesprungen. Insgesamt haben wir drei verschiedene Phasen gezeichnet. Warum aber dann vier Umschalt-Phasen? Ganz einfach: Nehmen wir als Beispiel den roten Punkt, der vom Spieler nicht berührt werden darf. Dieser Punkt soll in unserer Animation drei Phasen bekommen: Groß, mittel und klein. Hat man jetzt von mittel auf klein geschaltet, so käme als nächste Phase wieder groß an die Reihe. Dies würde aber einen Phasen-Sprung verursachen. Daher aktivieren wir als vierte Animations-Phase wieder die mittlere Größe. Dadurch ergibt sich eine scheinbar kontinuierliche Bewegung.

```
spriteanim:     add.b  #1,animcount ;Sprite Animationen
                cmp.b  #10,animcount
                beq.s  firstanim
                cmp.b  #20,animcount
                beq.s  secondanim
                cmp.b  #30,animcount
                beq.s  thirdanim
                cmp.b  #40,animcount
                beq    fourthanim
                rts
```

Nun folgen die einzelnen Unterroutinen, welche die einzelnen Phasen aus ihrer Speicherstelle holen und in den, am Bildschirm sichtbaren Bereich kopieren.

```
firstanim:      lea     se1,a1
                lea     se0,a0
                bsr     ani
                lea     sm1,a1
                lea     sm0,a0
                bsr     ani
                lea     sn1,a1
                lea     sn0,a0
                bsr     ani
                lea     sb1,a1
                bra     anim
secondanim:     lea     se2,a1
                lea     se0,a0
                bsr     ani
                lea     sm2,a1
                lea     sm0,a0
                bsr     ani
                lea     sn2,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb2,a1
                bra.s   anim
thirdanim:      lea     se3,a1
                lea     se0,a0
                bsr.s   ani
                lea     sm3,a1
                lea     sm0,a0
                bsr.s   ani
                lea     sn3,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb3,a1
                bra.s   anim
```

```
fourthanim:     clr.b   animcount
                add.b   #1,annis
                lea     se2,a1
                lea     se0,a0
                bsr.s   ani
                lea     sm2,a1
                lea     sm0,a0
                bsr.s   ani
                lea     sn2,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb2,a1
```

Damit das Listing nicht zu lange und kompliziert wird, wurde die, sich ständig wiederholende, Kopierschleife am Ende des Programms plaziert und jeweils mit bsr.s von oben angesprungen.

```
anim:           lea     sb0,a0
ani:            move    #15,d0
copyanim:       move.l (a1)+,(a0)+
                dbf     d0,copyanim
                rts
```

Es stellt sich abschließend die Frage wieviele Animations-Phasen nötig sind, um eine flüssige Bewegung zu realisieren. Wie Sie sehen, reichen die in unserem Fall verwendeten vier Phasen völlig aus. Aber je mehr Phasen Sie verwenden, desto schöner und sanfter bewegt sich Ihr Objekt - probieren Sie es!

3.3.4. ATTACHED SPRITES

Leider haben Sprites nur vier Farben, wovon eine Farbe transparent ist. Daher bleiben nur drei effektive Farben über. Mit einem kleinen Trick lassen sich jedoch 15 Farben (plus transparent) erzeugen. Der Amiga bietet uns an, zwei Sprites zu einem zu verbinden. Dadurch erhalten wir quasi vier Bitplanes, welche wiederum 16 Farben darstellen können (2 hoch 4 = 16).

Wir initialisieren alles wie gewöhnlich, nur daß wir zwei Sprites verwenden. Am besten zeichnen Sie Ihr Sprite mit einem Malprogramm und konvertieren es gleich als 15-farbiges Sprite. Somit erhalten Sie jeweils zwei Planes mit dem richtigen Farbmuster. Diese ergeben, übereinander gelegt, das 15 farbige Sprite. Diese müssen jedoch getrennt eingegeben werden, da sie auch so vom Amiga verwaltet werden und erst intern "übereinander gelegt" werden.

```
sprite2: dc.w $a08c,$b080 ;Buffer für Sprite1
```

Dies geschieht durch Setzen des sogenannten Attached-Bit, welches im zweiten Befehls-Wort des zweiten Sprites gesetzt wird. Schauen Sie sich dazu das folgende Listing an:

```
;
;  Darstellung von "attached" Sprites
;
;
s:
        move.w #$4000,$dff09a     ;Interrupts sperren;
                                  ;Betriebssystem ausschalten
        move   #$8020,$dff096     ;Sprites einschalten
        move.l #copperl,$dff084   ;Copperliste aktivieren
        move.l #pic,d0            ;Bild-Adresse in d0
        move   d0,plane1+6        ;Low word eintragen
        swap   d0
        move   d0,plane1+2        ;High word eintragen
        move.l #sprite1,d0        ;Sprite-Adresse in d0
        move   d0,sp1+6           ;Low word eintragen
        swap   d0
        move   d0,sp1+2           ;High word eintragen

        move.l #sprite2,d0        ;Sprite-Adresse in d0
        move   d0,sp1+14          ;Low word eintragen
        swap   d0
        move   d0,sp1+10          ;High word eintragen
;
loop:   move.l $dff004,d0         ;Warten auf Rasterstrahl
        and.l  #$fff00,d0
        cmp.l  #$00003000,d0
        bne.s  loop
        btst   #6,$bfe001         ;Maustaste gedrückt ?
        bne.s  loop               ;Ja ?
;
ende:   move   #$c000,$dff09a     ;Interrupts erlauben
e:      rts                       ;Programmende
;
copperl: dc.w $008e,$3081 ;DIWSTRT
         dc.w $0090,$35c1 ;DIWSTOP
         dc.w $0104,$0000 ;BPLCON2
         dc.w $0092,$0038 ;DDFSTRT
         dc.w $0094,$00d0 ;DDFSTOP
         dc.w $0102,$0000 ;BPLCON1
         dc.w $0108,$0000 ;BPL1MOD
         dc.w $010a,$0000 ;BPL2MOD
         dc.w $0100,$1200 ;BPLCON0
```

```
plane1:  dc.w $00e0,$0000 ;BPL1PTH
         dc.w $00e2,$0000 ;BPL1PTL
         dc.w $0180,$0000 ;COLOR00
         dc.w $0182,$0fff ;COLOR01
         dc.w $01a2,$0f00 ;COLOR17
         dc.w $01a4,$0c00 ;COLOR18
         dc.w $01a6,$0a00 ;COLOR19
         dc.w $01a8,$0800 ;COLOR20
         dc.w $01aa,$0500 ;COLOR21
         dc.w $01ac,$00f0 ;COLOR22
         dc.w $01ae,$00c0 ;COLOR23
         dc.w $01b0,$00a0 ;COLOR24
         dc.w $01b2,$0080 ;COLOR25
         dc.w $01b4,$0050 ;COLOR26
         dc.w $01b6,$000f ;COLOR27
         dc.w $01b8,$000c ;COLOR28
         dc.w $01ba,$000a ;COLOR29
         dc.w $01bc,$0008 ;COLOR30
         dc.w $01be,$0005 ;COLOR31
sp1:     dc.w $0120,$0000 ;SPR0PTH
         dc.w $0122,$0000 ;SPR0PTL
         dc.w $0124,$0000 ;SPR1PTH
         dc.w $0126,$0000 ;SPR1PTL
sp2:     dc.w $0128,$0000 ;SPR2PTH
         dc.w $012a,$0000 ;SPR2PTL
         dc.w $012c,$0000 ;SPR3PTH
         dc.w $012e,$0000 ;SPR3PTL
sp3:     dc.w $0130,$0000 ;SPR4PTH
         dc.w $0132,$0000 ;SPR4PTL
         dc.w $0134,$0000 ;SPR5PTH
         dc.w $0136,$0000 ;SPR5PTL
sp4:     dc.w $0138,$0000 ;SPR6PTH
         dc.w $013a,$0000 ;SPR6PTL
         dc.w $013c,$0000 ;SPR7PTH
         dc.w $013e,$0000 ;SPR7PTL
         dc.w $ffff,$fffe ;Warten auf unmögliche
                          ;Position
```

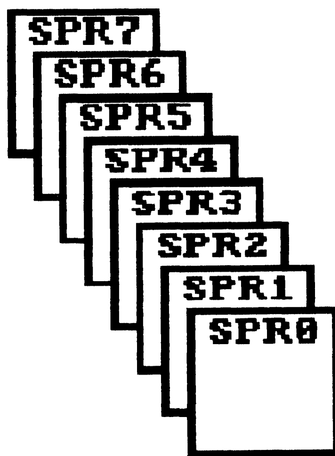
```
;
sprite1: dc.w $a08c,$b000 ;Buffer für Sprite0
         dc.w $FFFF,$0000
         dc.w $FFFF,$0000
         dc.w $0000,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $0000,$0000
         dc.w $FFFF,$0000
         dc.w $0000,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $0000,$0000
         dc.w $FFFF,$0000
         dc.w $0000,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $0000,$0000
         dc.w $FFFF,$0000
         dc.w $0000,$FFFF
         dc.w $FFFF,$FFFF

sprite2: dc.w $a08c,$b080 ;Buffer für Sprite1
         dc.w $0000,$0000 ;attached-Bit gesetzt
         dc.w $0000,$0000
         dc.w $0000,$0000
         dc.w $0000,$0000
         dc.w $FFFF,$0000
         dc.w $FFFF,$0000
         dc.w $FFFF,$0000
         dc.w $FFFF,$0000
         dc.w $0000,$FFFF
         dc.w $0000,$FFFF
         dc.w $0000,$FFFF
         dc.w $0000,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $FFFF,$FFFF
         dc.w $FFFF,$FFFF
;
pic:     blk.b 10240      ;Buffer für Bild
;
```

3.3.5. PRIORITÄTEN

Ein ganz wichtiger Punkt ist die Priorität der Grafik-Elemente untereinander. Unter Grafikelementen sind Playfields und Sprites zu verstehen, deren Reihenfolge wir beeinflußen können. Wie wir bereits wissen, gibt es sechs Bitplanes, aus denen wir entweder ein oder maximal zwei Playfields formen können. Haben wir den Dual-Playfield-Modus aktiviert, so ist normalerweise Playfield 1 vor Playfield 2 zu sehen. Setzt man aber das PF2PRI-Bit, so ist Playfield 2 im Vordergrund sichtbar.

Wie funktioniert das aber bei den Sprites? Nun, Sprites besitzen eine vorgegebene Reihenfolge, die nicht verändert werden kann. Man kann sich Sprites wie einen Stapel vorstellen, wobei das Sprite mit der niedrigsten Nummer als vorderstes liegt. Das zu hinterst befindliche Sprite ist nur sichtbar, wenn die im Vordergrund liegenden transparente Punkte enthalten, falls nicht bleibt es im Fall einer Überlappung verdeckt. Sehen wir uns das grafisch an:



Sprite 0 hat immer die höchste Priorität.

Die Priorität der Sprites ist nur in Bezug auf die/das Playfield/s wirksam. Man kann also nur Playfields zwischen Sprites positionieren, nicht aber die Reihenfolge der Sprites vertauschen.

Noch eine weitere Einschränkung erfahren die Sprites, denn sie werden, wie wir schon bei der Farbgebung gehört haben, in Zweier-Gruppen zusammengefaßt:

SPR 0&1, SPR 2&3, SPR 4&5, SPR 6&7

Daher ergeben sich nur fünf Möglichkeiten, zur Plazierung eines Playfields. Sehen Sie dazu die Grafik:

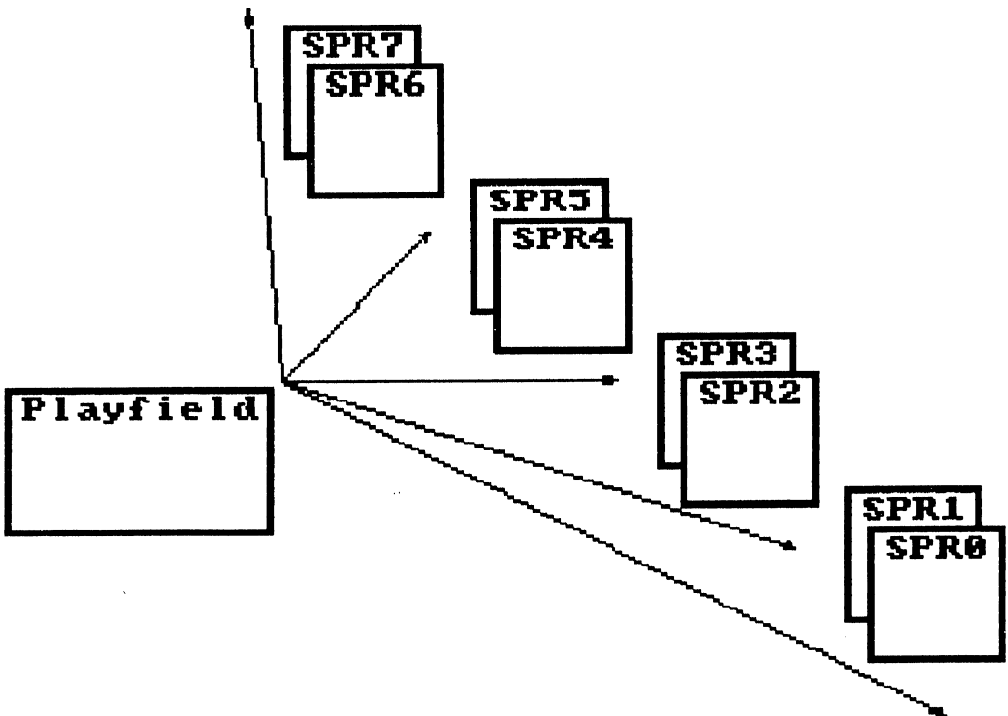


Bild 9: Playfield-Sprite Plazierung

Die Priorität wird im Register BPLCON2 ($dff104) festgelegt. Sehen wir uns die Bit-Belegung etwas genauer an:

BPLCON2 ($dff104)

Bit	Name	Funktion
15-7	---	unbenützt
6	PF2PRI	Dieses Bit bestimmt, welches Playfield im Vordergrund liegt. Ist dieses Bit gesetzt, hat Playfield 2 die Priorität vor Playfield 1.
5-3	PF2P2-PF2P0	Mit einer 3-Bit Kombination wird festgelegt, zwischen welchen Sprite-Paaren Playfield 2 liegt.
2-0	PF1P2-PF1P0	Wie PF2P2-PF2P0, nur gültig für Playfield 1.

Sehen wir uns nun an, welche 3-Bit Kombination verwendet werden muß, wenn man die Lage der Playfields zwischen den Sprites bestimmen will.

Bit-Wert	Plazierung				
000	PFx	SP01	SP23	SP45	SP67
001	SP01	PFx	SP23	SP45	SP67
010	SP01	SP23	PFx	SP45	SP67
011	SP01	SP23	SP45	PFx	SP67
100	SP01	SP23	SP45	SP67	PFx

Für unser Spiel setzen wir, da wir ein Dual-Playfield verwenden, Playfield 2 vor Playfield 1 und geben beiden Playfields die niedrigste Priorität, damit sämtliche Sprites davor sichtbar sind. Es sind folgende Bits zu setzen:

```
Bit 6 (PF2PRI)
Bit 5 (PF2P2) -> %100 = SP01 SP23 SP45 SP67 PF2
Bit 2 (PF1P2) -> %100 = SP01 SP23 SP45 SP67 PF1
-----------------------------------------------
```

Das Ergebnis: %01100100

Wir tragen diesen Wert hexadezimal in unsere Copperliste ein:

```
dc.w $0104,$0064
```

3.3.6. KOLLISIONEN

Ein weiterer, wichtiger Punkt, bei der Programmierung eines Spiels, ist die Erkennung von Kollisionen zwischen den bewegten Objekten. Dies können zum einen Kollisionen zwischen den Sprite-Paaren oder auch zwischen Playfields und Sprites sein. Des weiteren ist es möglich, im Dual-Playfield-Modus eine Kollision zwischen den beiden Playfields abzufragen. Dies ist dann besonders von Bedeutung, wenn man mit dem Blitter arbeitet und sogenannte Bobs verwendet. Man kann z.B. auf Playfield 1 den Hintergrund darstellen und auf Playfield 2 die Objekte "blitten". Will man eine Kollision zwischen einem Objekt (Bob) und dem Hintergrund abfragen, so muß man lediglich ein Bit testen.

Eine Kollision zwischen Sprites ist dann gegeben, wenn ein sichtbarer (nicht transparenter) Punkt des einen Sprites einen sichtbaren des anderen Sprites berührt (überlappt). Dieses Überlappen wird im CLXDAT-Register registriert und das entsprechende Bit gesetzt.

CLXDAT ($dff00e)

Bit	Funktion
15	unbelegt
14	Sprite 4 (oder 5) mit Sprite 6 (oder 7)
13	Sprite 2 (oder 3) mit Sprite 6 (oder 7)
12	Sprite 2 (oder 3) mit Sprite 4 (oder 5)
11	Sprite 0 (oder 1) mit Sprite 6 (oder 7)
10	Sprite 0 (oder 1) mit Sprite 4 (oder 5)
9	Sprite 0 (oder 1) mit Sprite 2 (oder 3)
8	Gerade Bitplanes (Playfield 2) mit Sprite 6 (oder 7)

7 Gerade Bitplanes (Playfield 2) mit Sprite 4 (oder 5)

6 Gerade Bitplanes (Playfield 2) mit Sprite 2 (oder 3)

5 Gerade Bitplanes (Playfield 2) mit Sprite 0 (oder 1)

4 Ungerade Bitplanes (Playfield 1) mit Sprite 6 (oder 7)

3 Ungerade Bitplanes (Playfield 1) mit Sprite 4 (oder 5)

2 Ungerade Bitplanes (Playfield 1) mit Sprite 2 (oder 3)

1 Ungerade Bitplanes (Playfield 1) mit Sprite 0 (oder 1)

0 Gerade Bitplanes (Playfield 2) mit ungeraden Bitplanes (Playfield 1)

Die ungeraden Sprites sind in Klammern gesetzt, weil diese nicht unbedingt zur Kollision hinzugezogen werden müssen. Wie wir wissen, sind die Sprites paarweise zugeordnet, leider auch bei der Kollisions-Abfrage. Man kann zwar wahlweise die ungeraden Sprites mit in die Abfrage einbeziehen, aber man kann mittels dieses Registers, im Falle einer Kollision, nicht unterscheiden, welches der beiden Sprites betroffen ist. Arbeitet man mit mehreren Sprites und ist eine Kollisions-Abfrage für diese unumgänglich und ist es darüberhinaus auch nicht möglich, die Objekte so zu plazieren, daß sie nur als gerade Sprites verwendet werden können, so empfiehlt es sich, z.B. zu einer Koordinaten-Abfrage zu greifen.

Für unser Spiel wollen wir das, vom Spieler gelenkte, Sprite (Sprite 0) auf Kollision mit dem roten Punkt (Sprite 2), dem grünen (Sprite 4) und dem gelben (Sprite 6) Objekt testen.

```
collision:      cmp.b   #$01,fin ;Kollisionsabfrage
                beq.s   nocolly
                move    $dff00e,d5
                btst    #9,d5
                bne     explusion
                btst    #10,d5
                bne     contact1
                btst    #11,d5
                bne     contact2
nocolly:        rts
```

Wird eine Kollsion durch das grüne Sprite verzeichnet, soll das violette Spieler-Symbol die Farbe auf Türkis wechseln. Dazu müssen wir folgende Routine schreiben:

```
contact1:       move.b #1,frage         ;Kollision mit grünem
                move   #$00fc,sprr0+6   ;Sprite
                move   #$00ca,sprr0+10
                move   #$00a8,sprr0+14
                rts
```

Hat der Spieler zuerst grün berührt und ist anschließend mit gelb zusammengestoßen, so soll er wieder die ursprüngliche Farbe erhalten und eine Nummer auf dem Anzeigenteil durchgestrichen werden. Weiters wird abgefragt, ob sich dieser Vorgang schon zehnmal wiederholt hat. Ist dies der Fall, wird das Spiel beendet.

```
contact2:       cmp.b  #1,frage         ;Kollision mit gelbem
                bne.s  nocontact        ;Sprite
                move   #$0f0c,sprr0+6
                move   #$0c0a,sprr0+10
                move   #$0a08,sprr0+14
                add.l  #2,kreuzptr
                clr.b  frage
                cmp.l  #18,kreuzptr
                bne.s  contdraw
                move   #$00f0,sieger+2
                move.b #1,fin
                move.b #1,end
                move.b #2,annis
                rts
contdraw:       bsr    drawfinis        ;Zahl durchstreichen
nocontact:      rts
```

Bei der Playfield Kollisions-Erkennung, läßt sich sogar genauer definieren, bei welcher Farbe (Bitkombination) eine Kollision ausgelöst werden soll. Sehen Sie dazu die Belegung des folgenden Registers:

CLXCON ($dff098)

Bit	Name	Funktion
15	ENSP7	Sprite 7 zur Kollision zulassen
14	ENSP5	Sprite 5 zur Kollision zulassen
13	ENSP3	Sprite 3 zur Kollision zulassen
12	ENSP1	Sprite 1 zur Kollision zulassen
11	ENBP6	Bitplane 6 zur Kollision zulassen (MVBP-Wert erforderlich!)
10	ENBP5	Bitplane 5 zur Kollision zulassen (MVBP-Wert erforderlich!)
9	ENBP4	Bitplane 4 zur Kollision zulassen (MVBP-Wert erforderlich!)
8	ENBP3	Bitplane 3 zur Kollision zulassen (MVBP-Wert erforderlich!)
7	ENBP2	Bitplane 2 zur Kollision zulassen (MVBP-Wert erforderlich!)
6	ENBP1	Bitplane 1 zur Kollision zulassen (MVBP-Wert erforderlich!)
5	MVBP6	Wert für Bitplane 6 Kollision
4	MVBP5	Wert für Bitplane 5 Kollision
3	MVBP4	Wert für Bitplane 4 Kollision
2	MVBP3	Wert für Bitplane 3 Kollision
1	MVBP2	Wert für Bitplane 2 Kollision
0	MVBP1	Wert für Bitplane 1 Kollision

Mit den Bits 15 - 12 lassen sich die ungeraden Sprites zu einer Kollisions-Erkennung einschalten. Gerade Sprites sind immer aktiviert.

Die Bits 11 - 6 bestimmen die, zur Kollision zugelassenen Bitplanes. Es werden nur die aktivierten Bitplanes verglichen.

Die restlichen Bits bestimmen, welches Bitmuster der jeweiligen Bitplane, die natürlich aktiviert werden muß, zur Kollision herangezogen werden soll. So ist es möglich, z.B. eine einzelne Farbe auf Kollision zu überprüfen.

In unserem Spiel soll sich der Spieler nur auf dem hinteren Playfield bewegen, wenn nicht, so soll seine Lebensenergie geringer werden. Dafür müssen wir folgende Bitkombination verwenden:

```
move #%0000000001000001,$dff098
```

Diese Initialisierung muß natürlich am Anfang des Programms geschehen. Innerhalb der Hauptschleife fügen wir noch die Abfrage der Kollision für die Bitplanes ein:

```
btst   #5,d5
beq    coll
btst   #1,d5
beq    coll
```

Jetzt fehlt noch die Unterroutine, die bestimmt, was geschehen soll, wenn eine Kollision stattgefunden hat.

```
coll:       add.b  #1,counter      ;Keine Kollision mit
            cmp.b  #10,counter     ;Bitplane
            bne.s  nosubtr
            clr.b  counter
            move.l powercount,a0
            clr.b  (a0)
            cmp.l  #pic3+1364,powercount
            beq.s  explosion
            add.l  #40,powercount
nosubtr:    rts
```

Mit dem "counter" wird festgelegt, wie schnell die Energie absinken soll. Anschließend wird eine Zeile (1 Byte breit) auf der Anzeigetafel gelöscht. Ein cmp-Befehl vergleicht, ob schon das Ende der Energie erreicht wurde, wenn ja wird zu "explosion" verzweigt.

3.4. SOUND-PROGRAMMIERUNG

Es gibt mehrere Möglichkeiten, Musik auf dem Amiga zu generieren. Wir wollen uns aber auf die einfachste Lösung beschränken, nämlich die einen Sample abzuspielen.

Diese Methode hat den Vorteil, daß sie sehr einfach zu programmieren ist. Da sie sehr viel Speicher kostet, kann man jedoch keine sehr langen Musikstücke abspielen. Ist der Sample abgespielt, wird wieder von vorne begonnen. Damit das Ganze nicht zu langweilig wird, kann man Abhilfe schaffen, indem man mehrere kurze Klänge digitalisiert und wie Noten abspielt. Da der Amiga vier Tonkanäle zur Verfügung stellt, ist es möglich vierstimmige Lieder mit Hilfe eines Editors zu komponieren. Die so erstellten Musikstücke benötigen natürlich viel weniger Speicherplatz und können minutenlang gespielt werden. Diese Musik zu programmieren ist ein ziemlich schwieriges Kapitel, und man könnte ein dickes Buch damit füllen. Das würde den Rahmen dieses Buches sprengen, daher wollen wir uns mit der einfachsten Methode befassen.

Um zu verstehen, wie der Amiga einen im Speicher liegenden Sample ausliest und abspielt, ist es nötig die verwendeten Register zu kennen:

```
AUD0LCH  ($dff0a0)  Zeiger auf Daten (high-word)
AUD0LCL  ($dff0a2)  Zeiger auf Daten (low-word)
```

Diese Register zeigen auf die, im Speicher liegenden, Musikdaten, die auf dem ersten Tonkanal abgespielt werden sollen. High- und low-word werden in zwei Registern getrennt verwaltet, aber wir können beide Register mit einem MOVE-Befehl initialisieren.

```
AUD1LCH  ($dff0b0)  Zeiger auf Daten (high-word)
AUD1LCL  ($dff0b2)  Zeiger auf Daten (low-word)
AUD2LCH  ($dff0c0)  Zeiger auf Daten (high-word)
AUD2LCL  ($dff0c2)  Zeiger auf Daten (low-word)
AUD3LCH  ($dff0d0)  Zeiger auf Daten (high-word)
AUD3LCL  ($dff0d2)  Zeiger auf Daten (low-word)
```

Die gleichen Register existieren natürlich analog zu den anderen drei Tonkanäle.

```
AUD0LEN  ($dff0a4)  Länge der Musik in Worten (Kanal 0)
AUD1LEN  ($dff0b4)  Länge der Musik in Worten (Kanal 1)
AUD2LEN  ($dff0c4)  Länge der Musik in Worten (Kanal 2)
AUD3LEN  ($dff0d4)  Länge der Musik in Worten (Kanal 3)
```

In diesen Register wird den vier Tonkanälen die Länge des, zu spielenden, Samples zugewiesen. Diese wird aber nicht in Bytes sondern in Words angegeben. Das bedeutet, die Länge des Musikstückes muß durch 2 geteilt werden.

```
AUD0PER  ($dff0a6)  Sampleperiode (Kanal 0)
AUD1PER  ($dff0b6)  Sampleperiode (Kanal 1)
AUD2PER  ($dff0c6)  Sampleperiode (Kanal 2)
AUD3PER  ($dff0d6)  Sampleperiode (Kanal 3)
```

Hier wird die Sampleperiode eingetragen. Diese Rate bestimmt, in welchen Abständen ein Datenbyte ausgegeben wird. Dadurch wird die Frequenz des zu spielenden Tons definiert.

```
AUD0VOL  ($dff0a8)  Lautstärke (Kanal 0)
AUD1VOL  ($dff0b8)  Lautstärke (Kanal 1)
AUD2VOL  ($dff0c8)  Lautstärke (Kanal 2)
AUD3VOL  ($dff0d8)  Lautstärke (Kanal 3)
```

Zuletzt wird die Lautstärke bestimmt. Diese kann einen Wert von 0 (aus) bis 64 (volle Lautstärke) annehmen. Der Vorteil liegt darin, daß jeder Tonkanal mit unterschiedlicher Lautstärke belegt werden kann, was zur Erzeugung der verschiedensten Effekte verwendet werden kann.

Nun können wir mit der Initialisierung unserer Musikroutine beginnen. Wir werden zwei der vier Tonkanäle verwenden. Die Musik liegt im Speicher unter dem Label "music".

```
initmusic:      move.l #music,$dff0a0
                move   #$9234,$dff0a4
                move   #200,$dff0a6
                move   #$40,$dff0a8

                move.l #music,$dff0b0
                move   #$9234,$dff0b4
                move   #200,$dff0b6
                move   #$40,$dff0b8
```

Das wäre schon alles, es fehlt nur noch das Einschalten selbst. Dazu sehen wir uns das Register DMACON ($dff096) an:

DMACON ($dff096)

Bit	Name	Funktion
15	SET/CLR	Das setzen/löschen control-Bit
14	BBUSY	Blitter status - testet ob der Blitter gerade arbeitet oder nicht (nur lesen).
13	BZERO	Blitter zero status - Das Ergebnis jeder Blitteroperation war 0 (nur lesen).
12	-	unbelegt
11	-	unbelegt
10	BLTPRI	Blitter priority - Der Blitter hat Vorrang gegenüber dem Prozessor.
9	DMAEN	DMA enable - Gesamtes DMA einschalten (gilt für Bits 0-8).
8	BPLEN	Bitplane DMA enable - Bitplane DMA einschalten
7	COPEN	Copper DMA enable - Copper DMA einschalten.
6	BLTEN	Blitter DMA enable - Blitter DMA einschalten.
5	SPREN	Sprite DMA enable - Sprite DMA einschalten.
4	DSKEN	Disk DMA enable - Disk DMA einschalten.
3	AUD3EN	Audio DMA channel 3 enable - Audiokanal 3 einschalten.
2	AUD2EN	Audio DMA channel 2 enable - Audiokanal 2 einschalten.
1	AUD1EN	Audio DMA channel 1 enable - Audiokanal 1 einschalten.

0 AUD0EN Audio DMA channel 0 enable - Audiokanal 0 einschalten.

Will man die Bits dieses Registers verändern, so muß man besonders auf das 15. Bit achten, da dieses entscheidet, ob die anderen Bits gesetzt oder gelöscht werden. Ist Bit 15 gelöscht, so werden alle anderen gesetzten Bits gelöscht. Ist das SET/CLR-Bit hingegen gesetzt, so werden alle anderen gesetzten Bits ebenfalls gesetzt. Verwirrend? Ein Beispiel verschafft Klarheit.

Um nun den ersten und den zweiten Kanal einzuschalten, müssen wir nicht nur die Bits für diese Kanäle setzen, sondern auch das control-Bit (Bit 15).

```
move #$8003,$dff096
```

Der Wert #$8003 ergibt sich aus der Bitkombination der gesetzten Bits 15,1 und 0. Man könnte natürlich auch wie folgt das Register initialisieren:

```
move #%1000000000000011,$dff096
```

Um die Musik wieder auszuschalten, setzen wir die Audio-Bits nocheinmal, aber da wir die Bits löschen wollen, muß auch das SET/CLR-Bit gelöscht werden.

```
move #$0003,$dff096
```

3.5. DER BLITTER

Neben dem Copper gehört der Blitter mit zu den wichtigsten Coprozessoren des Amigas. Das Wort Blitter stammt aus dem Englischen und ist eine Abkürzung für "Block Image Transferer", was auf Deutsch soviel wie "Bilddaten-Verschieber" heißt. Wie der Name schon vermuten läßt, ist es seine Hauptaufgabe, Daten im Speicher zu verschieben. Da er dies mit sehr großer Geschwindigkeit schafft, werden meistens Grafikdaten im Speicher kopiert. Daher hört man den Namen Blitter meistens in Zusammenhang mit Spielen und Grafikprogrammen. Tatsache ist aber, daß man mit dem Blitter auch die verschiedensten Verknüpfungen programmieren kann. Daher wird er oft zur Codierung und Decodierung von Daten, insbesondere von Disketten verwendet.

Weitere Anwendungsgebiete des Blitters sind das Zeichnen von Linien und das Füllen von Flächen. All diese Aufgaben kann der Blitter um einiges schneller erledigen, als der 68000er. Daher wird er sehr häufig in Spielen verwendet, um schnelle Animationen zu garantieren. Aber auch das Betriebssystem bedient sich seiner Hilfe um Fenster, Gadgets und Images zu bewegen oder richtig zu plazieren.

In unserem Spiel wird der Blitter nicht eingesetzt, weil wir weder große Datenblöcke kopieren, noch Zeit sparen müssen. Wir kopieren unsere Daten mit Hilfe des 68000er.
Der Vollständigkeit wegen, wollen wir in diesem Buch jedoch nicht auf den Blitter verzichten.

Die Anwendung des Blitters ist ziemlich einfach, und läuft jedesmal gleich ab. Es gibt drei verschiedene Datenquellen, die logisch miteinander verknüpft werden können. Das Ergebnis wird in die Zielquelle geschrieben, welche wir ebenfalls frei im Speicher wählen können. Diese Speicherbereiche, in denen unsere Quellen liegen, müssen sich im Chip-Memory befinden, damit der Blitter darauf zugreifen kann.

Um den Blitter, Daten kopieren zu lassen, müssen wir ihm vorher einige Informationen übermitteln, die er für seine Aufgabe benötigt:

- o Den gewünschten Modus (kopieren, füllen oder Linien ziehen anwählen).
- o Den oder die Quellbereiche bestimmen.
- o Den Zielbereich festlegen.
- o Verknüpfung und Maskierung auswählen.
- o Weitere Parameter definieren, wie zum Beispiel: Scrolling, Moduli, ...
- o Fenstergröße festlegen -> Blitter starten.

Der letzte Punkt, die Bestimmung der Fenstergröße, ist für den Blitter gleichzeitig das Signal zum Starten des Kopiervorganges. Daher muß man diese Funktion als letzte Initialisieren.

3.5.1. DIE BLITTERREGISTER

Um die oben erklärten Definitionen dem Blitter mitzuteilen, gibt es folgende Register:

BLTSIZE ($dff058)

Dieses Register legt nicht nur, wie der Name schon vermuten läßt, die Größe des Blitterfensters fest, sondern, es startet den Blitter auch gleich. Daher muß dieses Register als letztes initialisiert werden, da dem Blitter sonst noch nötige Informationen fehlen. Wie sieht aber das Blitterfenster aus? Das Blitterfenster ist jener Speicherbereich, der vom Blitter während der Operation verwendet wird. Eingeteilt wird der Speicher, genau wie bei den Bitplanes, in Spalten und Zeilen. Eine Spalte entspricht einem Word = zwei Byte.

Sehen wir uns nun die genauere Biteinteilung des BLTSIZE-Registers an:

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	H9	H8	H7	H6	H5	H4	H3	H2	H1	H0	W5	W4	W3	W2	W1	W0

Die ersten zehn Bits bestimmen die Höhe des Blitterfensters. Die Höhe wird in Zeilen angegeben, welche zwischen 0 und 2 hoch 10 (=1024) liegen kann. Will man 1024 Zeilen anwählen, so sind alle Bits auf Null zu setzen. Eine Höhe von Null Zeilen ist logischerweise nicht möglich.

Die restlichen sechs Bits stehen für die Breite, welche ebenfalls eine Größe von 0 bis 1024 Pixel haben darf. Die Breite wird in Words angegeben; 1 Word hat 16 Bits (=16 Pixel) - maximal 64 Words ergeben: 64*16=1024.
Es gilt hier die gleiche Regelung wie bei der Höhe: Will man 64 Words wählen, so sind diese Bits auf Null zu setzen. Eine Breite von Null Words ist unmöglich.

Um nun den richtigen Wert für BLTSIZE zu ermitteln, bedient man sich einer ganz einfachen Formel:

BLTSIZE = Höhe*64 + Breite in Words

BLTAPTH ($dff050)

Dieses Register enthält, zusammen mit dem Register BLTAPTL ($dff052), die Startadresse der Quelle A. Es existieren insgesamt vier Register (A, B, C und D). Davon sind drei (A, B und C) die Quellregister und eines (D) das Zielregister. Der Blitter holt die Informationen aus den entsprechenden Quellregistern, verknüpft sie wahlweise miteinander und schreibt das Ergebnis in das Zielregister D. In BLTAPTH liegen die 3 high Bits der Startadresse. Am Ende einer Blitteroperation liegt in diesen Registern die Adresse des letzten Words plus 2, plus dem entsprechenden Modulo Wert.

BLTAPTL ($dff052)

In diesem Register liegen die 15 low Bits der Startadresse von Quellregister A. Man kann beide Register mit einem einzigen Move Befehl beschreiben:

```
move.l  #pic,$dff052
```

BLTBPTH ($dff04c)

BLTBPTL ($dff04e)

BLTCPTH ($dff048)

BLTCPTL ($dff04a)

BLTDPTH ($dff054)

BLTDPTL ($dff056)

Die oben angeführten Register, sind genauso zu behandeln, wie BLTAPTH und BLTAPTL.

BLTAMOD ($dff064)

In diesem Register befindet sich der Modulo Wert für die Quelle A. Da der Blitter genau wie die Playfields auch mit rechteckigen Speicherbereichen arbeitet, müssen wir ihm einen Modulowert zuteilen. Die Funktionsweise der Modulo Werte ist im Kapitel 4.2 Playfields genau erklärt.

BLTBMOD ($dff062)

BLTCMOD ($dff060)

BLTDMOD ($dff066)

Modulo Werte gibt es sowohl für alle Quellregister, als auch für das Zielregister. Die Anwendung ist bei allen gleich.

BLTAFWM ($dff044)

Wie wir oben erfahren haben, läßt sich der Blitter nur wortweise steuern, was ist also zu tun, wenn man eine Grafik kopieren möchte, die über eine Wortgrenze hinausragt. Hier bietet uns der Blitter die Möglichkeit, die ungewünschten Bits der "überhängenden" Wörter auszumaskieren, sprich wegzulassen. Mit diesem Register wird das erste Word ausmaskiert. Die Maskierung funktioniert ziemlich einfach: Jedes auf 1 gesetzte Bit wird beim Kopiervorgang übernommen. Alle nicht gesetzten Bits werden gelöscht.

Ein Beispiel:

Nehmen wir an, unser Objekt besteht aus drei Words, wobei sowohl das erste als auch das letzte nur zur Hälfte übernommen werden, das mittlere hingegen ganz kopiert werden soll.

Vor der Operation:

Word 1	Word 2	Word 3
0000111111110000	0000111111110000	0000111111110000
0001111111111000	0001111111111000	0001111111111000
0011111111111100	0011111111111100	0011111111111100
0111111111111110	0111111111111110	0111111111111110
1111111111111111	1111111111111111	1111111111111111
0111111111111110	0111111111111110	0111111111111110
0011111111111100	0011111111111100	0011111111111100
0001111111111000	0001111111111000	0001111111111000
0000111111110000	0000111111110000	0000111111110000

Nach der Operation durch Verwendung folgender Masks:

BLTAFWM:		BLTALWM:
0000000011111111		1111111100000000
0000000011110000	0000111111110000	0000111100000000
0000000011111000	0001111111111000	0001111100000000
0000000011111100	0011111111111100	0011111100000000
0000000011111110	0111111111111110	0111111100000000
0000000011111111	1111111111111111	1111111100000000
0000000011111110	0111111111111110	0111111100000000
0000000011111100	0011111111111100	0011111100000000
0000000011111000	0001111111111000	0001111100000000
0000000011110000	0000111111110000	0000111100000000

BLTALWM ($dff046)

Dieses Register bestimmt die Maske des letzten Words. Sowohl BLTAFWM als auch BLTALWM sollten im Line- und im Fill-Modus auf eins gesetzt werden.

BLTCON0 ($dff040)

Bevor wir uns die Belegung dieses Registers ansehen, müssen wir zuerst noch einiges klären: Wir wissen, daß der Blitter wortweise arbeitet. Will man aber ein Objekt, das nicht an einer Wortgrenze endet kopieren, muß man, wie oben schon erläutert, das erste und das letzte Wort ausmaskieren. Was soll man aber tun, wenn man eine Grafik lediglich verschieben möchte?
Auch dafür stellt uns der Blitter eine Funktion zur Verfügung. Mit den Bits 12-15 wird bestimmt, um wieviele Bits verschoben werden soll. Die hinausgeschobenen Bits werden beim nächsten Word wieder hinein'geshiftet'. Sehen wir uns ein Beispiel an:

Vor dem Verschieben:

0001010111000000

Nach dem Verschieben:

xxxx000101011100

In unserem Beispiel wurde das Wort um vier Bits verschoben. Die "xxxx" repräsentieren jene Bits, die im vorhergehenden Wort stehen. Eine Verschiebung gibt es nur für Quelle A und B. Die Bits zur Steuerung von Quelle B befinden sich im Register BLTCON1.

Wie schon oben erwähnt, bietet uns der Blitter die Möglichkeit, die drei Quellregister logisch miteinander zu verknüpfen. Das Ergebnis dieser Verknüpfung wird in dem Zielregister D abgelegt. Uns stehen acht unterschiedliche Bit-Tripel zur Verfügung (2 hoch 3 = 8). Diese sind in den LFx-Bits (Bits 0-7) enthalten. Diese Bit-Tripel nennt man Miniterms. Mittels dieser Miniterms kann der Anwender bestimmen, wie das Ergebnis in D gebildet werden soll.

Sehen wir uns die Bit-Belegung des Registers an:

Bit	Name	Funktion
15	ASH3	Diese vier Bits beinhalten den Wert der Verschiebung von Quelle A. Sind alle 4 Bits auf Null gesetzt, so findet keine Verschiebung statt.
14	ASH2	
13	ASH1	
12	ASH0	
11	USEA	DMA-Kanal für Quelle A einschalten.
10	USEB	DMA-Kanal für Quelle B einschalten.
9	USEC	DMA-Kanal für Quelle C einschalten.
8	USED	DMA-Kanal für Ziel D einschalten.
7	LF7	Miniterm ABC (=111)
6	LF6	Miniterm ABc (=110)
5	LF5	Miniterm AbC (=101)
4	LF4	Miniterm Abc (=100)
3	LF3	Miniterm aBC (=011)
2	LF2	Miniterm aBc (=010)
1	LF1	Miniterm abC (=001)
0	LF0	Miniterm abc (=000)

BLTCON1 ($dff042)

Bit	Name	Funktion
15	BSH3	Diese vier Bits beinhalten den Wert der Verschiebung von Quelle B. Sind alle 4 Bits auf Null gesetzt, so findet keine Verschiebung statt.
14	BSH2	
13	BSH1	
12	BSH0	
11 - 5		unbenützt
4	EFE	Exclusive Fill Enable Mit den Bits 2, 3 und 4 sind nur in Zusammenhang mit der Funktion Flächen füllen interessant. Wollen Sie aber normal arbeiten, so sind beide Bits auf Null zu setzen.
3	IFE	Inclusive Fill Enable
2	FCI	Fill Carry In
1	DESC	Wenn dieses Bit eins ist, dann ist der decending Modus aktiviert. Wenn nicht, arbeitet der Blitter normal im ascending Modus. Was bedeuten nun diese beiden Modi? Normalerweise arbeitet der Blitter aufsteigend (ascending), das heißt, er beginnt mit dem Kopieren der Daten bei der Anfangsadresse und erhöht diese solange bis er bei der Endadresse angelangt ist. Will man aber einen Speicherbereich kopieren, wobei sich Quelle und Ziel teilweise Überlappen, so führt dies nicht zum gewünschten Ergebnis. Daher kann man den Blitter auch umgekehrt arbeiten lassen, nämlich absteigend (descending). Er beginnt jetzt bei der Endadresse und subtrahiert solange bis er bei der Startadresse angelangt ist.

0	LINE	Wird dieses Bit auf eins gesetzt, so ist der Line-Modus aktiviert und der Blitter kann zum Ziehen von Linien verwendet werden.

BLTADAT ($dff074)

BLTBDAT ($dff072)

BLTCDAT ($dff070)

BLTDDAT ($dff000)

Die Daten die kopiert werden sollen, werden vom Blitter, mit der Hilfe von vier DMA Kanälen, aus dem Speicher geholt, bearbeitet (verknüpft) und anschließend wieder zurück geschrieben. Nach der Verarbeitung steht das Ergebnis in BLTDDAT und wird danach ins RAM (Chip-Ram) geschrieben. Man kann jeden dieser Kanäle, mit den USEx-Bits des Registers BLTCON0 einzeln an- oder ausschalten. Will man einen Kanal sperren, so ist das entsprechende USEx-Bit auf Null zu setzen, sowie bei der Auswahl der Miniterme darauf zu achten, daß diese keinen Einfluß auf diesen Kanal nehmen.

Die folgenden Bits aus dem Register DMACON beeinflußen ebenfalls den Blitter.

DMACON ($dff096)

Bit	Name	Funktion
14	BBUSY	Blitter status - testet ob der Blitter gerade arbeitet oder nicht (nur lesen).
13	BZERO	Blitter zero status - Das Ergebnis jeder Blitteroperation war 0 (nur lesen).
10	BLTPRI	Blitter priority - Der Blitter hat Vorrang gegenüber dem Prozessor.
6	BLTEN	Blitter DMA enable - Blitter DMA einschalten.

3.5.2. EINFACHE BLITTEROPERATIONEN

Nach der langen Theorie wollen wir uns die Praxis, anhand von zwei Beispiel-Programmen, ansehen.

Wie man einen Speicherbereich mittels move-Befehl löscht, wissen wir bereits. Nun wollen wir dafür den Blitter zu Hilfe nehmen, der diese Aufgabe in viel kürzerer Zeit erledigt als der Prozessor. Sehen wir uns ein Beispielprogramm an, das ein Feld der Größe 320 mal 256 Punkten (320/8*256=10240 Byte) löscht. Diesen Speicherbereich definieren wir am Ende unseres Programms mit der Anweisung

```
pic:            blk.b  10240,$ff
```

$ff bedeutet, daß der reservierte Speicher beim Assemblieren mit dem Hex-Wert $ff gefüllt wird. Dadurch können wir nach dem Start kontrollieren, ob unser Programm richtig gearbeitet hat. Danach fragen wir den Blitter, ob er gerade beschäftigt ist:

```
clearpic:       btst   #14,$dff002
                bne.s  clearpic
```

Ist er gerade beim Kopieren von Daten, so verzweigt das Programm in die Schleife und wartet, bis er seine Aufgabe beendet hat. Der nächste Schritt ist das Initialisieren des Zielkanals.

```
                move.l #pic,$dff054
```

Außerdem löschen wir die Register, die wir für unsere Operation nicht benötigen. Dies sind die Moduli sowie das Register BLTCON1.

```
                clr    $dff066
                clr    $dff042
```

Jetzt müssen wir noch den Zielkanal einschalten:

```
                move   #%0000000100000000,$dff040
```

Abschließend wird die Größe des Blitterfensters festgelegt und der Blitter somit in Gang gesetzt.

```
                move   #[256*64]+[320/16],$dff058
```

Nun können wir das Programm starten.
Zur Überprüfung sehen wir uns den Speicher mit

```
                q.l pic
```

an. Es sollten ab dieser Adresse nur noch Nullen anstatt des Strings $ff stehen. Abschließend finden Sie das komplette Listing:

```
;
;       loescht eine 320x256 grosses Fenster
;       mit Hilfe des Blitters
;
clearpic:       btst    #14,$dff002
                bne.s   clearpic
;
                move.l #pic,$dff054
                clr     $dff066
                clr     $dff042
                move    #%0000000100000000,$dff040
                move    #[256*64]+[320/16],$dff058
;
                rts
;
pic:            blk.b   10240,$ff
;
```

Jetzt wollen wir eine Routine schreiben, die ein 32 farbiges Bob in einen Bildschirm, mit ebenfalls fünf Bitplanes, hineinkopiert. Wir werden jede Plane mit einem Blitvorgang duplizieren und dazu den Blitter fünfmal starten. Diese Methode benötigt viel Zeit, ist aber leichter zu verstehen. Dazu lesen wir den Anfang der ersten Plane in a1 ein und addieren nach jedem Blit $2800 hinzu, um an den Beginn der nächsten Plane zu gelangen. Der Wert aus a1 wird in das Zielregister übertragen:

```
label:          move.l a1,$dff054
```

Außerdem setzen wir alle nichtbenötigten Register auf Null.

```
blitten:        clr.w   $DFF042
                clr.w   $DFF064
```

Da wir weder das erste noch das letzte 'Word' ausmaskieren wollen, müssen wir sämtliche Bits auf eins setzen.

```
                move.w #$ffff,$dff044
                move.w #$ffff,$dff046
```

Darüberhinaus wird der Modulo-Wert für die Zielquelle angegeben, sowie der DMA-Kanal für Quelle A und Ziel D eingeschalten. Weiters werden alle Miniterms gewählt, welche bei A eine wahre Aussage liefern. Dies sind die Bits LF4 - LF7.

```
                move.w #%0000100111110000,$DFF040
                move.w #$24,$DFF066
```

Abschließend berechnen wir noch die Größe des Blitter-Fensters und starten es. Unser Bob ist 32 mal 25 Punkte groß, daher errechnet sich die Fenstergröße wie folgt aus der bekannten Formel:

BLTSIZE = Höhe*64 + Breite in Words

BLTSIZE = 25*64 + 2

BLTSIZE = 1602 ($642)

Sehen wir uns nun das fertige Programm an, das einen Screen mit 5 Bitplanes erstellt und ein 32 farbiges Bob hineinkopiert. Damit Sie etwas sehen, müssen Sie folgende auf der Beispieldiskette enthaltenen Bilder (Hintergrundbild und Bob) an die richtigen Positionen laden. Dies geschieht mit Hilfe des "ri" Kommandos (Seka-Assembler).

```
                spiel/1.con --> blitpic
                pictures/5planes.con --> pic
```

Hier nun das fertige Beispielprogramm:

```
;
;       Einfaches Kopieren mit dem Blitter
;
;
s:              bsr.s   opening
                bsr.L   blitten
;
;
loop:           move.l  $dff004,d0
                and.l   #$fff00,d0
                cmp.l   #$00003000,d0
                bne.s   loop
                btst    #6,$bfe001
                bne.s   loop
;
;
end:            move.w  #$c000,$dff09a
e:              rts
;
opening:        move.w  #$4000,$dff09a
                move.l  #copperl,$dff084
                move.w  #$20,$dff096
                move.l  #pic,d0
                move    d0,pl1+6
                swap    d0
                move    d0,pl1+2
                swap    d0
                add.l   size,d0
                move    d0,pl2+6
                swap    d0
                move    d0,pl2+2
                swap    d0
                add.l   size,d0
                move    d0,pl3+6
                swap    d0
                move    d0,pl3+2
                swap    d0
                add.l   size,d0
                move    d0,pl4+6
                swap    d0
                move    d0,pl4+2
```

```
                swap    d0
                add.l   size,d0
                move    d0,pl5+6
                swap    d0
                move    d0,pl5+2
                rts
;
                clr.w   $DFF042
                clr.w   $DFF064
;
                move.w #$ffff,$dff044
                move.w #$ffff,$dff046
;
                move.w #%0000100111110000,$DFF040
                move.w #$24,$DFF066
                move.l #blitpic,$DFF050
                lea     pic+2000,a1
                moveq   #4,d4
;
label:          move.l a1,$dff054
                add.l   #$2800,a1
                move.w #$642,$DFF058
;
test:           btst    #6,$DFF002
                bne.s   test
;
                dbf     d4,label
                rts
;
copperl:
                dc.w $008e,$2071
                dc.w $0090,$20d4
                dc.w $0092,$0038
                dc.w $0094,$00d0
                dc.w $0102,$0000
                dc.w $0104,$0000
                dc.w $0108,$0000
                dc.w $010a,$0000
                dc.w $0100,$5200
pl1:            dc.w $00e0,$0000
                dc.w $00e2,$0000
pl2:            dc.w $00e4,$0000
                dc.w $00e6,$0000
```

```
pl3:            dc.w $00e8,$0000
                dc.w $00ea,$0000
pl4:            dc.w $00ec,$0000
                dc.w $00ee,$0000
pl5:            dc.w $00f0,$0000
                dc.w $00f2,$0000
                dc.w $0180,$0000
                dc.w $0182,$0f00
                dc.w $0184,$00f0
                dc.w $0186,$000f
                dc.w $0188,$0444
                dc.w $018a,$0555
                dc.w $018c,$0666
                dc.w $018e,$0777
                dc.w $0190,$0888
                dc.w $0192,$0999
                dc.w $0194,$0aaa
                dc.w $0196,$0bbb
                dc.w $0198,$0ccc
                dc.w $019a,$0ddd
                dc.w $019c,$0eee
                dc.w $019e,$0fff
                dc.w $01a0,$0000
                dc.w $01a2,$0111
                dc.w $01a4,$0222
                dc.w $01a6,$0333
                dc.w $01a8,$0444
                dc.w $01aa,$0555
                dc.w $01ac,$0666
                dc.w $01ae,$0777
                dc.w $01b0,$0888
                dc.w $01b2,$0999
                dc.w $01b4,$0aaa
                dc.w $01b6,$0bbb
                dc.w $01b8,$0ccc
                dc.w $01ba,$0ddd
                dc.w $01bc,$0eee
                dc.w $01be,$0fff
                dc.w $ffff,$fffe
;
size:           dc.l   10240
pic:            blk.b  51200,0
blitpic:        blk.b  500,0
;
```

3.6. SONSTIGES

An dieser Stelle wollen wir uns jenen Dingen widmen, die man nicht so recht in eines der vorhergehenden Kapitel einordnen kann.

Außerdem programmieren wir noch einen kleinen Effekt, der mit dem Spiel selbst nicht viel zu tun hat, aber für einige Programmierer sicherlich interessant ist. Dabei handelt es sich um einen Sternen-Himmel, der mit möglichst einfachen Mitteln die optimalste Wirkung erzielt.

Abschließend besprechen wir kurz jene Schritte, die notwendig sind, um alle Einzelteile, die wir bis jetzt programmiert haben, zum gesamten Spiel zusammenzusetzen.

3.6.1. Joystick-Abfrage

Ein wesentlicher Bestandteil eines Spieles ist wohl die Steuerung, denn nichts nervt einen Spieler mehr, als eine schlechte Umsetzung der Bewegungen. Daher sollte man auf diesen Punkt sehr achten.

Es gibt mehrere Möglichkeiten den Joystickport auszulesen. Wir verwenden eine der kürzesten und schnellsten Methoden. Wir lesen das Joystick-Register $dff00c (JOY1DAT) in ein Datenregister ein und überprüfen den Inhalt. Wird nun der Joystick in eine Richtung bewegt, so stehen in d1 folgende Werte:

0003	rechts
0300	links
0100	oben
0001	unten
0200	links + oben
0002	rechts + unten
0103	rechts + oben
0301	links + unten

Wir müssen also lediglich mit cmp das Datenregister auf einen dieser Werte überprüfen, um bei erfolgreichem Vergleich in die gewünschte Unterroutine zu verzweigen. Für Port 0 (Mausport) ist folgendes Register zu verwenden:

JOY0DAT $dff00a

Die fertige Joystickabfrage für unser Spiel benötigt nur vier Bewegungsrichtungen. Wurde der Hebel gedrückt, verzweigt das Programm in die jeweilige Unterroutine und addiert bzw. subtrahiert 1 von der Spriteadresse. Darüberhinaus wird verglichen, ob das Sprite schon eine Randgrenze erreicht hat. Wenn ja wird nicht addiert/subtrahiert, sondern gleich zum Ende der Routine verzweigt.

```
;
; Die komplette Joystick-Abfrage
;
;
joy:        cmp.b  #$01,fin
            beq.s  joyout
            clr    $dff036
            move   $dff00c,d1
            cmp    #$0300,d1
            beq.s  links
            cmp.b  #$03,d1
            beq.s  rechts
            cmp    #$0100,d1
            beq.s  oben
            cmp.b  #$01,d1
            beq.s  unten
joyout:     rts
links:      move.l spritebuffer1,d0
            swap   d0
            cmp.b  #$40,d0
            bls.s  joyout
            sub    #$0001,spritebuffer1
            rts
rechts:     move.l spritebuffer1,d0
            swap   d0
            cmp.b  #$d8,d0
            bhs.s  joyout
            add    #$0001,spritebuffer1
            rts
oben:       move.l spritebuffer1,d0
            swap   d0
            clr.b  d0
            cmp    #$3000,d0
            bls.s  joyout
            sub.l  #$01000100,spritebuffer1
            rts
unten:      move.l spritebuffer1,d0
            swap   d0
            clr.b  d0
            cmp    #$ef00,d0
            bhs.s  joyout
            add.l  #$01000100,spritebuffer1
            rts
;
```

Ein weiterer Punkt ist das Auslesen der Feuerknöpfe. Die Abfrage des Joystickbuttons ist genau die gleiche, wie die der Maus:

```
button:     btst   #6,$bfe001
            bne.s  button
```

Die Abfrage für Port 1 liegt nur ein Bit weiter, nämlich:

```
button:     btst   #7,$bfe001
            bne.s  button
```

3.6.2. TASTATUR-ABFRAGE

Fast ebenso wichtig wie die Joystick-Abfrage ist die Auswertung der Tastatur. In unserem Spiel kommt dies zwar nicht zur Anwendung, trotzdem ist dieses Kapitel interessant.

Der Amiga besitzt ein intelligentes Keyboard, mit einem eigenen Prozessor, der automatisch die Auswertung übernimmt. Wir müssen lediglich den Puffer, in welchem die Daten gespeichert werden, auslesen.

Zuerst rettet man den Inhalt des Registers $bfec01, in dem sich die zuletzt gedrückte Taste befindet und vergleicht anschließend diesen mit den Tastatur-Codes.

```
loop:       move.b   $bfec01,d0
            cmp.b    #$41,d0
            bne.s    loop
            rts
```

Dieses kurze Programm wartet solange, bis die Help-Taste gedrückt wurde. Die Codes für die verschiedenen Tasten entnehmen Sie bitte der untenstehenden Abbildung. Diese entspricht der normalen Amiga 500 Tastatur. Es wurden aber auch schon andere Tastatur-Arten ausgeliefert z.B. für den Amiga 1000. Diese Tastaturen unterscheiden sich durch die Anzahl und Anordnung der Tasten. Es gibt Tastaturen, die um einige Tasten erweitert wurden. Wenn man obiges Programm so verändert, daß es den Wert der gerade gedrückten Zahl liefert, so kann man auch die Codes der nicht angeführten Tasten leicht ermitteln.

Wie Sie vielleicht bemerkt haben, entsprechen die angegebenen Codes nicht den ASCII-Codes. Legt man Wert darauf diese zu erhalten, so muß man eine Tabelle anlegen, die jedem Code das passende ASCII-Zeichen zuordnet.

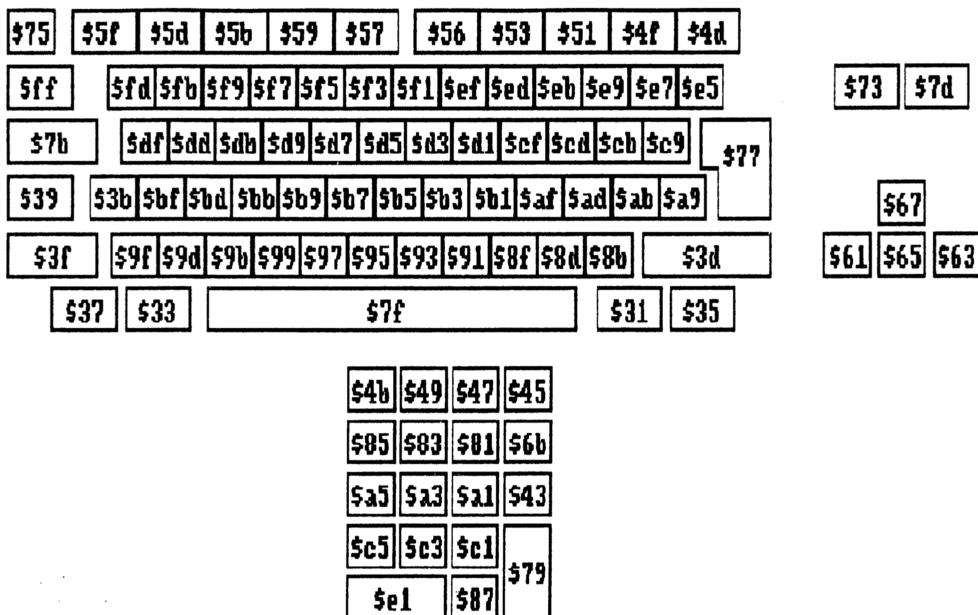


BILD 10: Die Tastaturbelegung eines Amiga 500

3.6.3. STERNENHIMMEL

Ein ebenfalls sehr häufig verwendeter Effekt ist das Sternen-Scrolling. Wir wollen diesen Effekt nachprogrammieren. Folgendes Ziel wollen wir uns dabei setzen:

- o 80 Sterne sollen gleichzeitig über den Bildschirm bewegt werden.
- o Sie sollen in unterschiedlichen Geschwindigkeiten aus der Bildschirmmitte dem Betrachter entgegenfliegen.

Diese Aufgabenstellung hört sich schwieriger an, als sie in Wirklichkeit ist. Beginnen wir einmal mit dem Einfachsten, dem Erstellen eines Bildschirms mit nur einer Plane. Da unsere Sterne einfarbig sind, genügt uns eine Bitplane.

Als nächstes legen wir eine Tabelle mit den Startkoordinaten für X und Y an. Diese Koordinaten geben die Ausgangspositionen unserer Sterne an:

```
skorsx:     dc.w 70,80,90,100,110,120,130,140,150,160
            dc.w 75,89,34,45,145,123,23,43,100,139
            ....

skorsy:     dc.w 55,60,65,70,75,80,85,90,95,100
            dc.w 56,76,64,45,67,98,65,54,76,86
            ....
```

Diese Positionen sind willkürlich im Koordinatensystem gewählt und können nach Belieben geändert werden. In jeder Zeile sind 10 Werte eingetragen, das bedeutet, daß noch 60 weitere fehlen, die wir, der Einfachkeit wegen, mit denselben Werten belegt haben.

Außerdem folgen zwei Tabellen, welche die Richtung und die Geschwindigkeit der Sterne innerhalb eines Quadranten festlegen.

```
speed1:     dc.w 1,2,3,4,5,6,2,4,5,6
            dc.w 6,4,0,1,3,6,1,7,3,3
            dc.w -1,-2,-3,-4,-5,-6,-2,-4,-5,-6
            dc.w -6,-4,-0,-1,-3,-6,-1,-7,-3,-3
            dc.w -1,-2,-3,-4,-5,-6,-2,-4,-5,-6
            dc.w -6,-4,-0,-1,-3,-6,-1,-7,-3,-3
            dc.w 1,2,3,4,5,6,2,4,5,6
            dc.w 6,4,0,1,3,6,1,7,3,3

speed2:     dc.w 1,1,1,1,1,1,1,1,1,1
            dc.w 1,1,1,1,1,1,0,0,1,1
            dc.w 1,1,1,1,1,1,1,1,1,1
            dc.w 1,1,1,1,1,1,0,0,1,1
            dc.w -1,-1,-1,-1,-1,-1,-1,-1,-1,-1
            dc.w -1,-1,-1,-1,-1,-1,0,0,-1,-1
            dc.w -1,-1,-1,-1,-1,-1,-1,-1,-1,-1
            dc.w -1,-1,-1,-1,-1,-1,0,0,-1,-1
```

Widmen wir uns als nächstes der eigentlichen Bewegungsroutine. Zuerst retten wir sämtliche Register, indem wir diese am Stack ablegen.

```
movestars:  movem.l d0-d7/a0-a6,-(a7)
```

Der nächste Schritt ist das Einlesen der Koordinaten, sowie der Geschwindigkeitsparameter. Weiters wird die Anzahl der Sterne in d7 übergeben.

```
            lea    skorsx(pc),a0
            lea    skorsy(pc),a1
            lea    speed1(pc),a2
            lea    speed2(pc),a3
            move.l #79,d7
```

Nachdem wir alle Parameter in den einzelnen Registern übergeben haben, können wir in der nächsten Routine mit der Berechnung der richtigen Bildschirm-Position beginnen. Die folgenden Zeilen erledigen diese Aufgabe für uns. Zuerst werden die Koordinaten, mit denen der äußersten Ränder, verglichen. Ist ein Stern über diese Koordinate bewegt worden, so wird er wieder auf die Ausgangsposition (x=160, y=128) zurückgesetzt. Dieser Vergleich muß natürlich für alle vier Bildschirmseiten durchgeführt werden.

```
starloop:   move    (a0),d0
            move    (a1),d1
            move    (a2)+,d2
            move    (a3)+,d3
            move    d0,d4
            move    d1,d5
            sub     d2,d0
            sub     d3,d1
            cmp     #370,d0
            ble.s   nox
            move    #160,d0
            move    #128,d1
            bra.s   noy
nox:        cmp     #9,d0
            bge.s   nomix
            move    #160,d0
            move    #128,d1
            bra.s   nomiy
nomix:      cmp     #270,d1
            ble.s   noy
            move    #160,d0
            move    #128,d1
            bra.s   nomiy
noy:        cmp     #9,d1
            bge.s   nomiy
            move    #160,d0
            move    #128,d1
nomiy:      move.w d0,(a0)
            move.w d1,(a1)+
```

Der letzte Schritt ist das Berechnen der Bildschirmposition, sowie das Löschen des zuletzt bewegten Sterns und das Setzen des neuen Punktes. Somit wäre unsere Routine für einen Stern komplett. Damit der Computer aber 80 Sterne bewegt, müssen wir am Ende mit einer dbf-Schleife diesen Vorgang weitere 79 Mal wiederholen. Abschließend werden die geretteten Register wieder zurückgeschrieben.

```
;
            mulu    #48,d5
            lsr.w   #3,d4
            lea     pic(pc),a5
            add     d4,a5
            add     d5,a5
            clr.b   (a5)
;
            lea     pic(pc),a5
            lsr.w   #3,d0
            mulu    #48,d1
            add     d0,a5
            add     d1,a5
;
            move.w (a0)+,d0
            and.w   #$7,d0
            move.l #128,d1
            lsr.l   d0,d1
            move.b d1,(a5)
;
            dbf     d7,starloop

            movem.l (a7)+,d0-d7/a0-a6
            rts
```

Die Stern-Routine hätten wir fertiggestellt. 80 Punkte fliegen von der Bildschirmmitte dem Betrachter quasi entgegen. Da sie in verschiedenen Geschwindigkeiten bewegt werden, wirkt unser kleines Weltall fast realistisch. Um diesen 3D-Effekt noch zu verstärken, werden wir den Sternen verschiedene Farben geben (Helligkeitsstufen), damit der Eindruck erweckt wird, als ob einige weiter im Hintergrund bewegt werden. Diesen Effekt werden wir ohne Zuhilfenahme einer weiteren Bitplane programmieren, sondern wir werden dazu den Copper verwenden. Wie schon aus der Skizze ersichtlich, verwenden wir zwei Copperlisten, wobei eine von oben nach unten und die andere von unten nach oben gescrollt wird. Zuerst müssen diese Listen initialisiert werden.

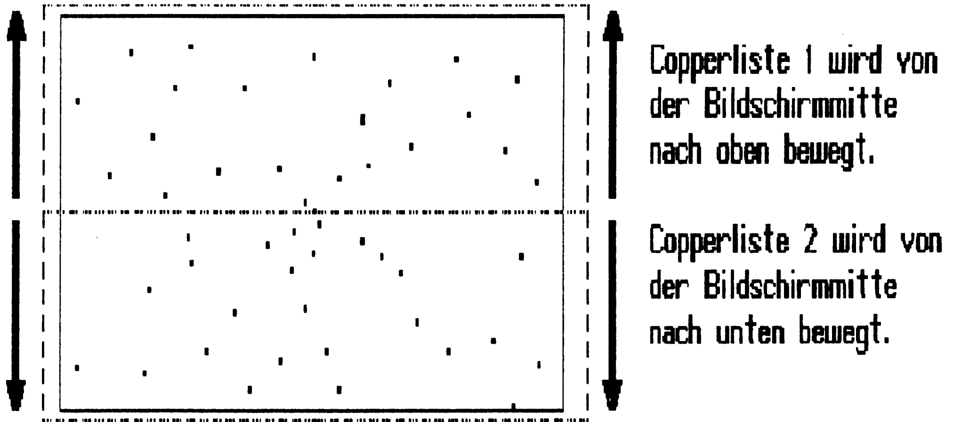


BILD 11: Der Sternenhimmel

```
;
initcopper: lea    cins(pc),a0
            move   #$0001,d0
            move   #159,d1
coppercopy: move   d0,(a0)+
            move   #$fffe,(a0)+
            move   #$0182,(a0)+
            add.w  #1,d2
            move   d2,(a0)+
            add    #$0100,d0
            dbf    d1,coppercopy
```

Die oben stehende Copperliste erstellt pro Bildschirmzeile einen Wait- und einen Farb-Move-Befehl. Die zweite Copperliste wird analog dazu installiert. Sehen wir uns als nächstes jene Routine an, welche die Farben der Copperliste scrollt:

```
;
scrollcopper1: lea cins+6(pc),a0
            move.l colorptr1(pc),a1
            cmp.w  #$ffff,(a1)
            bne.s  dascroll
            move.l #color1,colorptr1
            move.l colorptr1(pc),a1
dascroll:   move.w #159,d0
scrollen:   move.w (a1)+,(a0)
            cmp.w  #$ffff,(a1)
            bne.s  dascrollen
            lea    color1(pc),a1
dascrollen: add.l  #8,a0
            dbf    d0,scrollen
            add.l  #2,colorptr1
            rts
```

Zum Abschluß finden Sie das komplette Listing, das 80 Sterne darstellt und in verschiedenen Geschwindigkeiten und Helligkeitsstufen auf dem Bildschirm bewegt.

```
;
;
;           Sternenhimmel
;           =============
;
;
s:           bsr.s  opening
             bsr.l  initcopper
;
;
loop:        move.l $dff004,d0
             and.l  #$fff00,d0
             cmp.l  #$00003000,d0
             bne.s  loop
             bsr.l  movestars
             bsr    scrollcopper1
             bsr    scrollcopper2
             btst   #6,$bfe001
             bne.s  loop
;
;
end:         move.w #$c000,$dff09a
e:           rts
;
opening:     move.w #$4000,$dff09a
             move.l #copperl,$dff084
             move.w #$20,$dff096
             move.l #pic,d0
             move   d0,pl1+6
             swap   d0
             move   d0,pl1+2
             rts
;
copperl:
```

```
              dc.w $008e,$2071
              dc.w $0090,$20d4
              dc.w $0092,$0038
              dc.w $0094,$00d0
              dc.w $0102,$0000
              dc.w $0104,$0024
              dc.w $0108,$0008
              dc.w $010a,$0008
              dc.w $0100,$1200
pl1:          dc.w $00e0,$0000
              dc.w $00e2,$0000
              dc.w $0180,$0000
cins:         blk.w 640,0
cins2:        blk.w 640,0
              dc.w $ffff,$fffe
;
initcopper: lea     cins(pc),a0
              move    #$0001,d0
              move    #159,d1
coppercopy: move    d0,(a0)+
              move    #$fffe,(a0)+
              move    #$0182,(a0)+
              add.w   #1,d2
              move    d2,(a0)+
              add     #$0100,d0
              dbf     d1,coppercopy
;
              lea     cins2(pc),a0
              move    #$a001,d0
              move    #159,d1
coppercopy1: move   d0,(a0)+
              move    #$fffe,(a0)+
              move    #$0182,(a0)+
              add.w   #1,d2
              move    d2,(a0)+
              add     #$0100,d0
              dbf     d1,coppercopy1
              rts
;
```

```
scrollcopper1: lea cins+6(pc),a0
               move.l colorptr1(pc),a1
               cmp.w  #$ffff,(a1)
               bne.s  dascroll
               move.l #color1,colorptr1
               move.l colorptr1(pc),a1
dascroll:      move.w #159,d0
scrollen:      move.w (a1)+,(a0)
               cmp.w  #$ffff,(a1)
               bne.s  dascrollen
               lea    color1(pc),a1
dascrollen:    add.l  #8,a0
               dbf    d0,scrollen
               add.l  #2,colorptr1
               rts
;
scrollcopper2: lea cins2+1278(pc),a0
               move.l colorptr2(pc),a1
               cmp.w  #$ffff,(a1)
               bne.s  dascroll2
               move.l #color2,colorptr2
               move.l colorptr2(pc),a1
dascroll2:     move.w #159,d0
scrollen2:     move.w (a1)+,(a0)
               cmp.w  #$ffff,(a1)
               bne.s  dascrollen2
               lea    color2(pc),a1
dascrollen2: sub.l #8,a0
               dbf    d0,scrollen2
               add.l  #2,colorptr2
               rts
;
colorptr1:     dc.l   color1
color1:        dc.w $fff,$eee,$ddd,$ccc,$bbb,$aaa,$999,$888
               dc.w $777,$666,$555,$444,$333,$222,$111,$000
               dc.w $000,$111,$222,$333,$444,$555,$666,$777
               dc.w $888,$999,$aaa,$bbb,$ccc,$ddd,$eee,$fff
               dc.w $ffff
;
colorptr2:     dc.l   color2
color2:        dc.w $fff,$eee,$ddd,$ccc,$bbb,$aaa,$999,$888
               dc.w $777,$666,$555,$444,$333,$222,$111,$000
               dc.w $000,$111,$222,$333,$444,$555,$666,$777
               dc.w $888,$999,$aaa,$bbb,$ccc,$ddd,$eee,$fff
               dc.w $ffff
;
```

```
movestars:  movem.l d0-d7/a0-a6,-(a7)
            lea     skorsx(pc),a0
            lea     skorsy(pc),a1
            lea     speed1(pc),a2
            lea     speed2(pc),a3
            move.l  #79,d7
starloop:   move    (a0),d0
            move    (a1),d1
            move    (a2)+,d2
            move    (a3)+,d3
            move    d0,d4
            move    d1,d5
            sub     d2,d0
            sub     d3,d1
            cmp     #370,d0
            ble.s   nox
            move    #160,d0
            move    #128,d1
            bra.s   noy
nox:        cmp     #9,d0
            bge.s   nomix
            move    #160,d0
            move    #128,d1
            bra.s   nomiy
nomix:      cmp     #270,d1
            ble.s   noy
            move    #160,d0
            move    #128,d1
            bra.s   nomiy
noy:        cmp     #9,d1
            bge.s   nomiy
            move    #160,d0
            move    #128,d1
nomiy:      move.w  d0,(a0)
            move.w  d1,(a1)+
;
            mulu    #48,d5
            lsr.w   #3,d4
            lea     pic(pc),a5
            add     d4,a5
            add     d5,a5
            clr.b   (a5)
;
```

```
            lea     pic(pc),a5
            lsr.w   #3,d0
            mulu    #48,d1
            add     d0,a5
            add     d1,a5
;
            move.w (a0)+,d0
            and.w   #$7,d0
            move.l #128,d1
            lsr.l   d0,d1
            move.b d1,(a5)
;
            dbf     d7,starloop

            movem.l (a7)+,d0-d7/a0-a6
            rts
;
speed1:     dc.w 1,2,3,4,5,6,2,4,5,6
            dc.w 6,4,0,1,3,6,1,7,3,3

            dc.w -1,-2,-3,-4,-5,-6,-2,-4,-5,-6
            dc.w -6,-4,-0,-1,-3,-6,-1,-7,-3,-3

            dc.w -1,-2,-3,-4,-5,-6,-2,-4,-5,-6
            dc.w -6,-4,-0,-1,-3,-6,-1,-7,-3,-3

            dc.w 1,2,3,4,5,6,2,4,5,6
            dc.w 6,4,0,1,3,6,1,7,3,3

speed2:     dc.w 1,1,1,1,1,1,1,1,1,1
            dc.w 1,1,1,1,1,1,0,0,1,1

            dc.w 1,1,1,1,1,1,1,1,1,1
            dc.w 1,1,1,1,1,1,0,0,1,1

            dc.w -1,-1,-1,-1,-1,-1,-1,-1,-1,-1
            dc.w -1,-1,-1,-1,-1,-1,0,0,-1,-1

            dc.w -1,-1,-1,-1,-1,-1,-1,-1,-1,-1
            dc.w -1,-1,-1,-1,-1,-1,0,0,-1,-1
```

```
skorsx:     dc.w 70,80,90,100,110,120,130,140,150,160
            dc.w 75,89,34,45,145,123,23,43,100,139

            dc.w 70,80,90,100,110,120,130,140,150,160
            dc.w 75,89,34,45,145,123,23,43,100,139

            dc.w 70,80,90,100,110,120,130,140,150,160
            dc.w 75,89,34,45,145,123,23,43,100,139

            dc.w 70,80,90,100,110,120,130,140,150,160
            dc.w 75,89,34,45,145,123,23,43,100,139

skorsy:     dc.w 55,60,65,70,75,80,85,90,95,100
            dc.w 56,76,64,45,67,98,65,54,76,86

            dc.w 55,60,65,70,75,80,85,90,95,100
            dc.w 56,76,64,45,67,98,65,54,76,86

            dc.w 55,60,65,70,75,80,85,90,95,100
            dc.w 56,76,64,45,67,98,65,54,76,86

            dc.w 55,60,65,70,75,80,85,90,95,100
            dc.w 56,76,64,45,67,98,65,54,76,86

pic:        blk.b 12240,0
```

3.6.4. ANREGUNGEN

Zum Abschluß dieses Buches soll Ihnen dieses Kapitel weitere Anregungen zum Thema Spieleprogrammierungen, speziell zu unseren Beispielprogrammen, geben. Auf den folgenden Seiten sind einige Vorschläge zur Verbesserung, beziehungsweise zur Erweiterung der Programme aufgeführt. Mit den bis jetzt erworbenen Kenntnissen, müßten Sie in der Lage sein, die folgenden Änderungen vorzunehmen.

1) Zwei- oder Drei-Spielermodus

Unser kleines Spiel ist nur für einen Spieler, der gegen den Computer antritt, ausgelegt. Sie können nun einen zweiten Spieler mit einem in Port 0 angeschlossenen Joystick programmieren. Dazu müssen Sie lediglich die Routinen für die Joystick-Abfrage von Port 1 kopieren und auf Port 0 umändern, indem Sie das Register $dff00a verwenden.

Sogar ein dritter Spieler könnte in das Spielgeschehen eingreifen, wenn Sie seine Bewegungen über die Tastatur einlesen. Außerdem könnte man den Spieler selbst wählen lassen, mit welchen Tasten er sein Symbol lenken möchte, indem man ihn zu Beginn die Tasten selbst definieren läßt.

Spielen mehrere Personen gleichzeitig, so müssen natürlich auch die Kollisionen darauf eingerichtet werden.

2) Sternenhimmel Titelbild

Das Beispiel-Listing Sternenhimmel steht mit unserem Spiel nicht in Zusammenhang. Man könnte sich diesen netten Effekt aber als Titelvorspann zunutze machen. Nachdem man das Spiel gestartet hat, erscheint nicht gleich das Spielfeld, sondern ein schwarzer Bildschirm aus dessen Tiefe Sterne erscheinen. Außerdem könnte man den Titel in den Hintergrund einblenden. Um das zu erreichen, müssen Sie eine zweite Copperliste anlegen. Die Copperliste, die zum Hauptprogramm gehört, wird erst nach dem Druck und durch den Joystick aktiviert, dadurch schaltet der Computer zwischen Sternen und Spiel um. Wenn man die Sterne ebenfalls im Dual-Playfield-Modus programmiert, könnte man im zweiten Playfield den Titel des Spieles einblenden. Im Dual-Playfield-Modus kann man maximal 3 Bitplanes verwenden, das entspricht 8 Farben.

3) Attached Sprites

Da wir nur vier Sprites für unser Spiel verwenden, ist der Einsatz von Attached Sprites für alle vier Symbole möglich. Beachten müssen Sie allerdings, daß auch die Bits für die ungeraden Sprites bei der Kollision gesetzt werden müssen, damit diese überhaupt zugelassen werden (Register $dff098).

4) Soundeffekte

Im Kapitel Sound-Programmierung wurde die Anwendung und das Abspielen von Samples besprochen. Während Ablauf des Programms ertönt ein Sample als Hintergrund-Musik. Zur Steigerung, können Sie mit einem Soundsampler mehrere Klänge und Geräusche samplen und z.B. während einer Kollision abspielen. Ein netter Effekt ist auch eine gesampelte Stimme, die bei einer Kollision gespielt wird.

4) Blitter-Einsatz

Im Spiel wird der Blitter nicht verwendet, da keine großen Datenmengen kopiert werden müssen. Ein kleines Datenfeld wird allerdings kopiert, nämlich das Datenfeld, in dem die Zahl der Leben steht. Diese Aufgabe wird noch vom Prozessor übernommen. Im Beispielprogramm des Kapitels "Einfache Blitteroperationen" wird dieser Vorgang mit dem Blitter demonstriert. Ersetzen Sie diese Routinen auch im Spiel!

ANHANG

Im Anhang finden Sie eine, für Hardware-Programmierer unerläßliche Übersicht über alle Hardware-Register des Amiga. Außerdem ist an dieser Stelle das fertige Spiele-Listing, sowie ein Stichwort- und Literaturverzeichnis aufgeführt.

1. ÜBERSICHT DER HARDWARE-REGISTER

Am wichtigsten für Spiele-Programmierer sind die Hardware-Register. In diesem Kapitel finden Sie eine Auflistung aller Hardware-Register des Amiga.

NAME	ADRESSE	FUNKTION
BLTDDAT	$dff000	Blitter Destination Data
DMACONR	$dff002	DMA Control Read
VPOSR	$dff004	Vertical Position Read
VHPOSR	$dff006	Vert. and horiz. Position Read
DSKDATR	$dff008	Disk data read
JOY0DAT	$dff00a	Joy-Port 0 data
JOY1DAT	$dff00c	Joy-Port 1 data
CLXDAT	$dff00e	Collision data
ADKCONR	$dff010	Audio Disk Control Read
POT0DAT	$dff012	Pot0 Dat a
POT1DAT	$dff014	Pot1 Dat a
POTGOR	$dff016	Pot data read
SERDATR	$dff018	Serial data read
DSKBYTR	$dff01a	Disk data byte read
INTENAR	$dff01c	Interrupt enable bits read
INTREQR	$dff01e	Interrupt request bits read
DSKPTH	$dff020	Disk pointer high
DSKPTL	$dff022	Disk pointer low
DSKLEN	$dff024	Disk data length
DSKDAT	$dff026	Disk DMA data write
REFPTR	$dff028	Refresh pointer
VPOSW	$dff02a	Vertical Position write
VHPOSW	$dff02c	Vert. and horiz. Pos. write
COPCON	$dff02e	Coprozessor control register
SERDAT	$dff030	Serial port data
SERPER	$dff032	Serial port period
POTGO	$dff034	Pot port data write and go
JOYTEST	$dff036	Joy port Test
STREQU	$dff038	Strobe for horiz. sync with VB and EQU
STRVBL	$dff03a	Strobe for horiz. sync with VB
STRHOR	$dff03c	Strobe for horiz. sync
STRLONG	$dff03e	Strobe for identification of long horiz. line

BLTCON0	$dff040	Blitter control register 0
BLTCON1	$dff042	Blitter control register 1
BLTAFWM	$dff044	Blitter first word mask for A
BLTALWM	$dff046	Blitter last word mask for A
BLTCPTH	$dff048	Blitter pointer high to C
BLTCPTL	$dff04a	Blitter pointer low to C
BLTBPTH	$dff04c	Blitter pointer high to B
BLTBPTL	$dff04e	Blitter pointer low to B
BLTAPTH	$dff050	Blitter pointer high to A
BLTAPTL	$dff052	Blitter pointer low to A
BLTDPTH	$dff054	Blitter pointer high to D
BLTDPTL	$dff056	Blitter pointer low to D
BLTSIZE	$dff058	Blitter size and start
	$dff05a	nicht belegt
	$dff05c	nicht belegt
	$dff05e	nicht belegt
BLTCMOD	$dff060	Blitter modulo C
BLTBMOD	$dff062	Blitter modulo B
BLTAMOD	$dff064	Blitter modulo A
BLTDMOD	$dff066	Blitter modulo D
	$dff068	nicht belegt
	$dff06a	nicht belegt
	$dff06c	nicht belegt
	$dff06e	nicht belegt
BLTCDAT	$dff070	Blitter C data register
BLTBDAT	$dff072	Blitter B data register
BLTADAT	$dff074	Blitter A data register
	$dff076	nicht belegt
	$dff078	nicht belegt
	$dff07a	nicht belegt
	$dff07c	nicht belegt
DSKSYNC	$dff07e	Disk synchronisation pattern
COP1LCH	$dff080	Copper first location high
COP1LCL	$dff082	Copper first location low
COP2LCH	$dff084	Copper second location high
COP2LCL	$dff086	Copper second location low
COPJMP1	$dff088	Restart Copper first location
COPJMP2	$dff08a	Restart Copper sec. location
COPINS	$dff08c	Copper instruction fetch
DIWSTRT	$dff08e	Display window start
DIWSTOP	$dff090	Display window stop
DDFSTRT	$dff092	Display data fetch start
DDFSTOP	$dff094	Display data fetch stop
DMACON	$dff096	DMA control register

CLXCON	$dff098	Collision control
INTENA	$dff09a	Interrupt enable bits
INTREQ	$dff09c	Interrupt request bits
ADKCON	$dff09e	Audio and disk control
AUD0LCH	$dff0a0	Audio0 location high
AUD0LCL	$dff0a2	Audio0 location low
AUD0LEN	$dff0a4	Audio0 data length
AUD0PER	$dff0a6	Audio0 period
AUD0VOL	$dff0a8	Audio0 volume
AUD0DAT	$dff0aa	Audio0 data register
	$dff0ac	nicht belegt
	$dff0ae	nicht belegt
AUD1LCH	$dff0b0	Audio1 location high
AUD1LCL	$dff0b2	Audio1 location low
AUD1LEN	$dff0b4	Audio1 data length
AUD1PER	$dff0b6	Audio1 period
AUD1VOL	$dff0b8	Audio1 volume
AUD1DAT	$dff0ba	Audio1 data register
	$dff0bc	nicht belegt
	$dff0be	nicht belegt
AUD2LCH	$dff0c0	Audio2 location high
AUD2LCL	$dff0c2	Audio2 location low
AUD2LEN	$dff0c4	Audio2 data length
AUD2PER	$dff0c6	Audio2 period
AUD2VOL	$dff0c8	Audio2 volume
AUD2DAT	$dff0ca	Audio2 data register
	$dff0cc	nicht belegt
	$dff0ce	nicht belegt
AUD3LCH	$dff0d0	Audio3 location high
AUD3LCL	$dff0d2	Audio3 location low
AUD3LEN	$dff0d4	Audio3 data length
AUD3PER	$dff0d6	Audio3 period
AUD3VOL	$dff0d8	Audio3 volume
AUD3DAT	$dff0da	Audio3 data register
	$dff0dc	nicht belegt
	$dff0de	nicht belegt
BPL1PTH	$dff0e0	Bitplane1 pointer high
BPL1PTL	$dff0e2	Bitplane1 pointer low
BPL2PTH	$dff0e4	Bitplane2 pointer high
BPL2PTL	$dff0e6	Bitplane2 pointer low
BPL3PTH	$dff0e8	Bitplane3 pointer high
BPL3PTL	$dff0ea	Bitplane3 pointer low
BPL4PTH	$dff0ec	Bitplane4 pointer high
BPL4PTL	$dff0ee	Bitplane4 pointer low
BPL5PTH	$dff0f0	Bitplane5 pointer high

BPL5PTL	$dff0f2	Bitplane5 pointer low
BPL6PTH	$dff0f4	Bitplane6 pointer high
BPL6PTL	$dff0f6	Bitplane6 pointer low
	$dff0f8	nicht belegt
	$dff0fa	nicht belegt
	$dff0fc	nicht belegt
	$dff0fe	nicht belegt
BPLCON0	$dff100	Bitplane control register 0
BPLCON1	$dff102	Bitplane control register 1
BPLCON2	$dff104	Bitplane control register 2
	$dff106	nicht belegt
BPL1MOD	$dff108	Bitplane Modulo 1
BPL2MOD	$dff10a	Bitplane Modulo 2
	$dff10c	nicht belegt
	$dff10e	nicht belegt
BPL1DAT	$dff110	Bitplane1 data
BPL2DAT	$dff112	Bitplane2 data
BPL3DAT	$dff114	Bitplane3 data
BPL4DAT	$dff116	Bitplane4 data
BPL5DAT	$dff118	Bitplane5 data
BPL6DAT	$dff11a	Bitplane6 data
	$dff11c	nicht belegt
	$dff11e	nicht belegt
SPR0PTH	$dff120	Sprite0 pointer high
SPR0PTL	$dff122	Sprite0 pointer low
SPR1PTH	$dff124	Sprite1 pointer high
SPR1PTL	$dff126	Sprite1 pointer low
SPR2PTH	$dff128	Sprite2 pointer high
SPR2PTL	$dff12a	Sprite2 pointer low
SPR3PTH	$dff12c	Sprite3 pointer high
SPR3PTL	$dff12e	Sprite3 pointer low
SPR4PTH	$dff130	Sprite4 pointer high
SPR4PTL	$dff132	Sprite4 pointer low
SPR5PTH	$dff134	Sprite5 pointer high
SPR5PTL	$dff136	Sprite5 pointer low
SPR6PTH	$dff138	Sprite6 pointer high
SPR6PTL	$dff13a	Sprite6 pointer low
SPR7PTH	$dff13c	Sprite7 pointer high
SPR7PTL	$dff13e	Sprite7 pointer low
SPR0POS	$dff140	Sprite0 start position
SPR0CTL	$dff142	Sprite0 control data
SPR0DATA	$dff144	Sprite0 image data A
SPR0DATB	$dff146	Sprite0 image data B
SPR1POS	$dff148	Sprite1 start position

SPR1CTL	$dff14a	Sprite1 control data
SPR1DATA	$dff14c	Sprite1 image data A
SPR1DATB	$dff14e	Sprite1 image data B
SPR2POS	$dff150	Sprite2 start position
SPR2CTL	$dff152	Sprite2 control data
SPR2DATA	$dff154	Sprite2 image data A
SPR2DATB	$dff156	Sprite2 image data B
SPR3POS	$dff158	Sprite3 start position
SPR3CTL	$dff15a	Sprite3 control data
SPR3DATA	$dff15c	Sprite3 image data A
SPR3DATB	$dff15e	Sprite3 image data B
SPR4POS	$dff160	Sprite4 start position
SPR4CTL	$dff162	Sprite4 control data
SPR4DATA	$dff164	Sprite4 image data A
SPR4DATB	$dff166	Sprite4 image data B
SPR5POS	$dff168	Sprite5 start position
SPR5CTL	$dff16a	Sprite5 control data
SPR5DATA	$dff16c	Sprite5 image data A
SPR5DATB	$dff16e	Sprite5 image data B
SPR6POS	$dff170	Sprite6 start position
SPR6CTL	$dff172	Sprite6 control data
SPR6DATA	$dff174	Sprite6 image data A
SPR6DATB	$dff176	Sprite6 image data B
SPR7POS	$dff178	Sprite7 start position
SPR7CTL	$dff17a	Sprite7 control data
SPR7DATA	$dff17c	Sprite7 image data A
SPR7DATB	$dff17e	Sprite7 image data B
COLOR00	$dff180	Color register 00
COLOR01	$dff182	Color register 01
COLOR02	$dff184	Color register 02
COLOR03	$dff186	Color register 03
COLOR04	$dff188	Color register 04
COLOR05	$dff18a	Color register 05
COLOR06	$dff18c	Color register 06
COLOR07	$dff18e	Color register 07
COLOR08	$dff190	Color register 08
COLOR09	$dff192	Color register 09
COLOR10	$dff194	Color register 10
COLOR11	$dff196	Color register 11
COLOR12	$dff198	Color register 12
COLOR13	$dff19a	Color register 13
COLOR14	$dff19c	Color register 14

COLOR15	$dff19e	Color register 15
COLOR16	$dff1a0	Color register 16
COLOR17	$dff1a2	Color register 17
COLOR18	$dff1a4	Color register 18
COLOR19	$dff1a6	Color register 19
COLOR20	$dff1a8	Color register 20
COLOR21	$dff1aa	Color register 21
COLOR22	$dff1ac	Color register 22
COLOR23	$dff1ae	Color register 23
COLOR24	$dff1b0	Color register 24
COLOR25	$dff1b2	Color register 25
COLOR26	$dff1b4	Color register 26
COLOR27	$dff1b6	Color register 27
COLOR28	$dff1b8	Color register 28
COLOR29	$dff1ba	Color register 29
COLOR30	$dff1bc	Color register 30
COLOR31	$dff1be	Color register 31

2. DAS KOMPLETTE SPIELE-LISTING

Zusammenfassend finden Sie hier das Listing für unser kleines Geschicklichkeits-Spiel. Das Programm wurde auf einem Seka-Assembler geschrieben, ist aber zu anderen Assemblern kompatibel. Evtl. müssen einige Befehle oder Adressierunggsarten verändert werden.
Das gesamte Spiel finden Sie natürlich auch auf der mitgelieferten Diskette, sowohl als Source-Listing wie auch als ausführbares Object-File. Ebenso sind alle notwendigen Grafik- und Musik-Daten auf der Programmdiskette enthalten.

```
;
;          %%%%%%%%%%%%%%%%%%%%%%
;          %%     THE GAME     %%
; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; %% Program, sound and graphics written %%
; %%           by Niki Laber             %%
; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;
s:              bsr    opening  ;Alles öffnen
                bsr    initcopper ;Copperliste erstellen
                bsr    initmusic ;Musik initialisieren
;
restart:        move.l #pic3+364,powercount
                clr.b  end
                clr.b  fin
                clr.b  animcount
                clr.b  frage
                move.l #-2,kreuzptr
                bsr    clr
                move.b #3,life
                bsr    drawpower ;Energiebalken zeichnen
                bsr    drawlife ;"Leben" zeichnen
                bsr    clrsprites ;Sprites löschen
                bsr    initsprites ;Sprites kopieren
                move   #$f77,aa1+2
                move   #$733,aa3+2
warten:         bsr    coloris  ;Farben
                bsr    wait  ;Warteschleife
                move   $dff00e,d5
                btst   #7,$bfe001 ;warte auf Joy-button
                bne.s  warten  ;gedrückt?
                move.l #$a08cb000,spritebuffer1
```

```
                move    #$c55,aa1+2
                move    #$c55,aa2+2
                move    #$c55,aa3+2
;
; Hauptschleife
;
loop:           move.l $dff004,d0 ;Warten auf Rasterzeile
                and.l   #$fff00,d0
                cmp.l   #$00003000,d0
                bne.s   loop
                bsr     joy  ;Joystick Abfrage
                bsr     scrolling_kl ;Scrolling kleine Plane
                bsr     scrolling_gr ;Scrolling große Plane
                bsr     collision ;Kollisions-check
                bsr     scrollcopper ;3D-Rolle
                bsr     moveenemy ;Feind bewegen
                bsr     move1  ;Symbol 1 bewegen
                bsr     move2  ;Symbol 2 bewegen
                bsr     color  ;Farben
                bsr.L   spriteanim ;Sprites animieren
                cmp.b   #1,fin
                bne.s   mouse
                cmp.b   #3,annis
                beq.s   nextlife
mouse:          btst    #6,$bfe001 ;Maustaste gedrückt?
                bne.s   loop  ;ja?
;
; Programmende
;
ende:           move    #$0003,$dff096 ;Sound abschalten
                move    #$c000,$dff09a ;Copper zurückschreiben
e:              rts                    ;Programmende
;
;
; Unterroutinen
;
;
opening:        move.w #$4000,$dff09a ;Interrupts sperren
                move.l  #copperl,$dff084;Copperliste aktivie-
ren
                move    #%0000000001000001,$dff098
;                                      Kollision ein
                move.l #pic1,d0 ;Playfield 1
                move    d0,pl1+6
                swap    d0
                move    d0,pl1+2
                move.l #pic1+gr1,d0
                move    d0,pl2+6
```

```
swap    d0
move    d0,pl2+2
move.l  #pic1+[gr1*2],d0
move    d0,pl3+6
swap    d0
move    d0,pl3+2

move.l  #pic2,d0 ;Playfield 2
move    d0,pl11+6
swap    d0
move    d0,pl11+2
move.l  #pic2+gr2,d0
move    d0,pl22+6
swap    d0
move    d0,pl22+2
move.l  #pic2+[gr2*2],d0
move    d0,pl33+6
swap    d0
move    d0,pl33+2

move.l  #pic3,d0 ;Anzeigetafel
move    d0,pl111+6
swap    d0
move    d0,pl111+2
move.l  #pic3+gr3,d0
move    d0,pl111+14
swap    d0
move    d0,pl111+10
move.l  #pic3+[gr3*2],d0
move    d0,pl111+22
swap    d0
move    d0,pl111+18
move.l  #pic3+[gr3*3],d0
move    d0,pl111+30
swap    d0
move    d0,pl111+26
move.l  #pic3+[gr3*4],d0
move    d0,pl111+38
swap    d0
move    d0,pl111+34

move.l  #spritebuffer1,d0 ;Sprites
move    d0,p1+6
swap    d0
move    d0,p1+2
move.l  #sprite3,d0
move    d0,p2+6
swap    d0
```

```
                move    d0,p2+2
                move.l #sprite4,d0
                move    d0,p3+6
                swap    d0
                move    d0,p3+2
                move.l #sprite5,d0
                move    d0,p4+6
                swap    d0
                move    d0,p4+2
                rts
;
nextlife:       cmp.b   #1,end
                beq.L   restart
                clr.b   fin
                clr.b   frage
                clr.b   animcount
                move    $dff00e,d5
                move.l #pic3+364,powercount
                bsr     drawpower
                bsr     initsprites
                move.l #$a08cb000,spritebuffer1
                move    #$c55,aa1+2
                move    #$c55,aa2+2
                move    #$c55,aa3+2
                bra.L   loop
;
initcopper:     lea     cins(pc),a0 ;Copperliste erstellen
                move.l #29,d0
                move.w #$c001,d1
copycop:        move.w d1,(a0)+
                move.w #$fffe,(a0)+
                move.w #$0180,(a0)+
                move.w #$0000,(a0)+
                add.w   #$0100,d1
                dbra    d0,copycop
                rts
;
spriteanim:     add.b   #1,animcount ;Sprite Animationen
                cmp.b   #10,animcount
                beq.s   firstanim
                cmp.b   #20,animcount
                beq.s   secondanim
                cmp.b   #30,animcount
                beq.s   thirdanim
                cmp.b   #40,animcount
                beq     fourthanim
                rts
firstanim:      lea     se1,a1
```

```
                lea     se0,a0
                bsr     ani
                lea     sm1,a1
                lea     sm0,a0
                bsr     ani
                lea     sn1,a1
                lea     sn0,a0
                bsr     ani
                lea     sb1,a1
                bra     anim
secondanim:     lea     se2,a1
                lea     se0,a0
                bsr     ani
                lea     sm2,a1
                lea     sm0,a0
                bsr     ani
                lea     sn2,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb2,a1
                bra.s   anim
thirdanim:      lea     se3,a1
                lea     se0,a0
                bsr.s   ani
                lea     sm3,a1
                lea     sm0,a0
                bsr.s   ani
                lea     sn3,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb3,a1
                bra.s   anim
fourthanim:     clr.b   animcount
                add.b   #1,annis
                lea     se2,a1
                lea     se0,a0
                bsr.s   ani
                lea     sm2,a1
                lea     sm0,a0
                bsr.s   ani
                lea     sn2,a1
                lea     sn0,a0
                bsr.s   ani
                lea     sb2,a1
anim:           lea     sb0,a0
ani:            move    #15,d0
copyanim:       move.l  (a1)+,(a0)+
                dbf     d0,copyanim
```

```
                rts
;
wait:           move    #$5000,d0 ;Warteschleife
waiting:        dbf     d0,waiting
                rts
;
collision:      cmp.b   #$01,fin ;Kollisionsabfrage
                beq.s   nocolly
                move    $dff00e,d5
                btst    #5,d5
                beq     coll
                btst    #1,d5
                beq     coll
                btst    #9,d5
                bne     explusion
                btst    #10,d5
                bne     contact1
                btst    #11,d5
                bne     contact2
nocolly:        rts
contact1:       move.b #1,frage ;Kollision mit grünem
                move    #$00fc,sprr0+6 ;Sprite
                move    #$00ca,sprr0+10
                move    #$00a8,sprr0+14
                rts
contact2:       cmp.b   #1,frage ;Kollision mit gelbem
                bne.s   nocontact ;Sprite
                move    #$0f0c,sprr0+6
                move    #$0c0a,sprr0+10
                move    #$0a08,sprr0+14
                add.l   #2,kreuzptr
                clr.b   frage
                cmp.l   #18,kreuzptr
                bne.s   contdraw
                move    #$00f0,sieger+2
                move.b #1,fin
                move.b #1,end
                move.b #2,annis
                rts
contdraw:       bsr     drawfinis
nocontact:      rts
coll:           add.b   #1,counter ;Keine Kollision mit
                cmp.b   #10,counter ;Bitplane
                bne.s   nosubtr
                clr.b   counter
                move.l powercount,a0
                clr.b   (a0)
                cmp.l   #pic3+1364,powercount
```

```
                beq.s  explusion
                add.l  #40,powercount
nosubtr:        rts
;
explusion:      move.b #1,fin   ;Explusion
                clr.b  annis
                sub.b  #1,life
                cmp.b  #0,life
                bne.s  da
                move.b #1,end
da:             cmp.b  #2,life
                bne.s  da1
                lea    life2,a1
                bsr    makelife
da1:            cmp.b  #1,life
                bne.s  da2
                lea    life1,a1
                bsr    makelife
da2:            lea    sb1(pc),a0
                lea    ss1(pc),a1
                move   #$0f60,sprr0+6
                move   #$0e40,sprr0+10
                move   #$0c00,sprr0+14
spritecopy1:    move   #47,d0
copyspr1:       move.l (a1)+,(a0)+
                dbf    d0,copyspr1
                rts
;
clrsprites:     clr.l  spritebuffer1 ;Sprites löschen
                clr.l  sprite2
                clr.l  sprite3
                clr.l  sprite4
                clr.l  sprite5
                rts
;
initsprites:    lea    sb1(pc),a0 ;Sprites kopieren
                lea    se1(pc),a1
                bsr.s  spritecopy1
                move   #$0f0c,sprr0+6
                move   #$0c0a,sprr0+10
                move   #$0a08,sprr0+14
                lea    sb1(pc),a0
                lea    s1(pc),a1
                bra.s  spritecopy1
;
moveenemy:      move.b l5,d4  ;Rotes Sprite bewegen
                move.b l6,d5
                add.b  d4,l3
```

```
               subq.b #1,l1
               bne.s  label1
               move.b #151,l1
               eor.b  #$fe,d4
label1:        move.b l3,sprite3+1
               add.b  d5,l4
               subq.b #1,l2
               bne.s  label2
               move.b #194,l2
               eor.b  #$fe,d5
label2:        move.b l4,sprite3
               move.b l4,d6
               add.b  #$10,d6
               move.b d6,sprite3+2
               move.b d4,l5
               move.b d5,l6
               rts
;
move1:         move.b l15,d4  ;Grünes Sprite bewegen
               move.b l16,d5
               add.b  d4,l13
               subq.b #2,l11
               bne.s  label11
               move.b #151,l11
               eor.b  #$fe,d4
label11:       move.b l13,sprite4+1
               add.b  d5,l14
               subq.b #1,l12
               bne.s  label12
               move.b #194,l12
               eor.b  #$fe,d5
label12:       move.b l14,sprite4
               move.b l14,d6
               add.b  #$10,d6
               move.b d6,sprite4+2
               move.b d4,l15
               move.b d5,l16
               rts
;
move2:         move.b l115,d4  ;Gelbes Sprite bewegen
               move.b l116,d5
               add.b  d4,l113
               subq.b #3,l111
               bne.s  label111
               move.b #151,l111
               eor.b  #$fe,d4
label111:      move.b l113,sprite5+1
               add.b  d5,l114
```

```
                subq.b #2,lll2
                bne.s  labellll2
                move.b #194,lll2
                eor.b  #$fe,d5
labellll2:      move.b lll4,sprite5
                move.b lll4,d6
                add.b  #$10,d6
                move.b d6,sprite5+2
                move.b d4,lll5
                move.b d5,lll6
                rts
;
scrolling_kl:   tst.b  kennbyte1 ;Scrolling des kleinen
                beq.s  rauf  ;Playfields
                lea    pl1+2(pc),a1 ;(Hintergrund)
                move   (a1),d0
                swap   d0
                move   4(a1),d0
                add.l  #40,d0
                move   d0,4(a1)
                swap   d0
                move   d0,(a1)

                lea    pl2+2(pc),a1
                move   (a1),d0
                swap   d0
                move   4(a1),d0
                add.l  #40,d0
                move   d0,4(a1)
                swap   d0
                move   d0,(a1)

                lea    pl3+2(pc),a1
                move   (a1),d0
                swap   d0
                move   4(a1),d0
                add.l  #40,d0
                move   d0,4(a1)
                swap   d0
                move   d0,(a1)
                add    #1,count1
                cmp    #145,count1
                bne.s  noscroll
                clr.b  kennbyte1
rauf:           add    #1,count1
                lea    pl1+2(pc),a1
                move   (a1),d0
                swap   d0
```

```
                move    4(a1),d0
                sub.l   #40,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)

                lea     pl2+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #40,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)

                lea     pl3+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #40,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
                cmp     #290,count1
                bne.s   noscroll
                clr     count1
                move.b  #$01,kennbyte1
noscroll:       rts
;
scrolling_gr:   tst.b   kennbyte2 ;Scrolling des großen
                beq.s   rauf2  ;Playfields
                lea     pl11+2(pc),a1 ;(Vordergrund)
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)

                lea     pl22+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
```

```
                lea     pl33+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                add.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
                add     #1,count2
                cmp     #173,count2
                bne.s   noscroll2
                clr.b   kennbyte2
rauf2:          add     #1,count2
                lea     pl11+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)

                lea     pl22+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)

                lea     pl33+2(pc),a1
                move    (a1),d0
                swap    d0
                move    4(a1),d0
                sub.l   #80,d0
                move    d0,4(a1)
                swap    d0
                move    d0,(a1)
                cmp     #346,count2
                bne.s   noscroll2
                clr     count2
                move.b  #$01,kennbyte2
noscroll2:      rts
;
scrollcopper:   lea     cins(pc),a0 ;3D-Rolle drehen und
                move.l parptr,a4 ;bewegen
```

```
                cmp.w   #$ffff,(a4)
                bne.s   tres
                move.l  #parabel,parptr
                move.l  parptr(pc),a4
tres:           move.l  shadeptr(pc),a3
                cmp.w   #$ffff,(a3)
                bne.s   qwex
                move.l  #shade,shadeptr
                move.l  shadeptr(pc),a3
qwex:           move.l  colorptr(pc),a2
                cmp.w   #$ffff,(a2)
                bne.s   ewewx
                move.l  #colors,colorptr
                move.l  colorptr(pc),a2
ewewx:          move    #29,d0
                move.w  (a4),d5
iix:            move    d5,(a0)
                add     #$0100,d5
                add.l   #6,a0
                move    (a2)+,d7
                move    (a3)+,d6
                and.w   d6,d7
                move    d7,(a0)
                cmp.w   #$ffff,(a2)
                bne.s   werwerx
                lea     colors(pc),a2
werwerx:        cmp.w   #$ffff,(a3)
                bne.s   trex
                lea     shade(pc),a3
trex:           add.l   #2,a0
                dbra    d0,iix
                add.l   #2,colorptr
                add.l   #2,parptr
                rts
;
joy:            cmp.b   #$01,fin ;Joystickabfrage
                beq.s   joyout
                clr     $dff036
                move    $dff00c,d1
                cmp     #$0300,d1
                beq.s   links
                cmp.b   #$03,d1
                beq.s   rechts
                cmp     #$0100,d1
                beq.s   oben
                cmp.b   #$01,d1
                beq.s   unten
joyout:         rts
```

```
links:          move.l spritebuffer1,d0
                swap   d0
                cmp.b  #$40,d0
                bls.s  joyout
                sub    #$0001,spritebuffer1
                rts
rechts:         move.l spritebuffer1,d0
                swap   d0
                cmp.b  #$d8,d0
                bhs.s  joyout
                add    #$0001,spritebuffer1
                rts
oben:           move.l spritebuffer1,d0
                swap   d0
                clr.b  d0
                cmp    #$3000,d0
                bls.s  joyout
                sub.l  #$01000100,spritebuffer1
                rts
unten:          move.l spritebuffer1,d0
                swap   d0
                clr.b  d0
                cmp    #$ef00,d0
                bhs.s  joyout
                add.l  #$01000100,spritebuffer1
                rts
;
initmusic:      move.l #music,$dff0a0 ;Musik initialisieren
                move   #$9234,$dff0a4 ;und einschalten
                move   #200,$dff0a6
                move   #$40,$dff0a8
                move.l #music,$dff0b0
                move   #$9234,$dff0b4
                move   #200,$dff0b6
                move   #$40,$dff0b8
                move   #$8003,$dff096
                rts
;
drawlife:       lea    life3,a1 ;restliche "Leben"
makelife:       lea    pic3+370,a2 ;kopieren
                move   #4,d1
planecopy1:     move.l a2,a0
                move   #24,d0
lifecopy1:      move.l (a1)+,(a0)
                add.l  #40,a0
                dbf    d0,lifecopy1
                add.l  #gr3,a2
                dbf    d1,planecopy1
```

```
                rts
;
drawfinis:      lea     pic3+456+[gr3*3],a0 ;Kreuze zeichnen
                add.l   kreuzptr,a0
                move.b  #%11000001,(a0)
                move.b  #%01100011,40(a0)
                move.b  #%00110110,80(a0)
                move.b  #%00011100,120(a0)
                move.b  #%00011100,160(a0)
                move.b  #%00110110,200(a0)
                move.b  #%01100011,240(a0)
                move.b  #%11000001,280(a0)
                rts
;
clr:            lea     pic3+456+[gr3*3],a0 ;Kreuze löschen
                move.l  #8,d0
                lea     pic3+456+[gr3*3],a0
kreuzen:        bsr.s   kreuz
                add.l   #2,a0
                dbf     d0,kreuzen
                rts
kreuz:          clr.b   (a0)
                clr.b   40(a0)
                clr.b   80(a0)
                clr.b   120(a0)
                clr.b   160(a0)
                clr.b   200(a0)
                clr.b   240(a0)
                clr.b   280(a0)
                rts
;
drawpower:      move.l powercount,a0 ;Energie zeichnen
                move    #24,d0
draw:           move.b #$ff,(a0)
                add.l   #40,a0
                dbf     d0,draw
                rts
;
color:          lea     sprr1+10,a0 ;Rotes Sprite
                move.l colptr,a1 ;"blinken" lassen
                cmp     #$ffff,(a1)
                bne.s   dahier
                move.l #colo,colptr
                move.l colptr,a1
dahier:         move    (a1),(a0)
                add.l   #2,colptr
                rts

;
```

```
coloris:        lea    aa2+2,a0 ;Schrift
                move.l colptr2,a1 ;"blinken" lassen
                cmp    #$ffff,(a1)
                bne.s  dahieris
                move.l #colo2,colptr2
                move.l colptr2,a1
dahieris:       move   (a1),(a0)
                add.l  #2,colptr2
                rts
;
; Definitionen
;
animcount: dc.b 0
kennbyte1: dc.b 1
kennbyte2: dc.b 1
l1:        dc.b 151
l2:        dc.b 194
l3:        dc.b 64
l4:        dc.b 44
l5:        dc.b 1
l6:        dc.b 1
ll1:       dc.b 151
ll2:       dc.b 194
ll3:       dc.b 54
ll4:       dc.b 34
ll5:       dc.b 1
ll6:       dc.b 1
lll1:      dc.b 151
lll2:      dc.b 194
lll3:      dc.b 54
lll4:      dc.b 34
lll5:      dc.b 1
lll6:      dc.b 1
counter:   dc.b 0
fin:       dc.b 0
annis:     dc.b 0
life:      dc.b 3
frage:     dc.b 0
end:       dc.b 0,0
count1:    dc.w 0
count2:    dc.w 0
kreuzptr:  dc.l 0
powercount:dc.l 0
;
colptr:  dc.l colo
colo: dc.w $0f00,$0e00,$0d00,$0c00,$0b00,$0a00,$0900,$0800
 dc.w $0700,$0600,$0500,$0400
 dc.w $0500,$0600,$0700,$0800,$0900,$0a00,$0b00,$0c00
```

```
 dc.w $0d00,$0e00,$0f00
 dc.w $ffff
;
colptr2: dc.l colo2
colo2:
 dc.w $0f00,$0f00,$0e00,$0e00,$0d00,$0d00,$0c00,$0c00
 dc.w $0b00,$0b00,$0a00,$0a00,$0900,$0900,$0800,$0800
 dc.w $0700,$0700,$0600,$0600,$0500,$0500,$0400,$0400
 dc.w $0300,$0300,$0200,$0200,$0100,$0100,$0000,$0000
 dc.w $0100,$0100,$0200,$0200,$0300,$0300,$0400,$0400
 dc.w $0500,$0500,$0600,$0600,$0700,$0700,$0800,$0800
 dc.w $0900,$0900,$0a00,$0a00,$0b00,$0b00,$0c00,$0c00
 dc.w $0d00,$0d00,$0e00,$0e00,$0f00,$0f00
 dc.w $ffff
;
colorptr: dc.l colors
colors:
dc.w $0f00,$0f00,$0f00,$0f00,$0f00
dc.w $0f00,$0f00,$0f00,$0f00,$0f00
dc.w $0fff,$0fff,$0fff,$0fff,$0fff
dc.w $0fff,$0fff,$0fff,$0fff,$0fff
dc.w $0f00,$0f00,$0f00,$0f00,$0f00
dc.w $0f00,$0f00,$0f00,$0f00,$0f00
dc.w $ffff
;
shadeptr: dc.l shade
shade:
dc.w $111,$222,$333,$444,$555,$666,$777,$888,$999,$aaa
dc.w $bbb,$ccc,$ddd,$eee,$fff,$fff,$eee,$ddd,$ccc,$bbb
dc.w $aaa,$999,$888,$777,$666,$555,$444,$333,$222,$111,$ffff
;
parptr: dc.l parabel
parabel:
dc.w $3001,$3001,$3001,$3001,$3201,$3201,$3201,$3301,$3301
dc.w $3301,$3501,$3501,$3501,$3601,$3601,$3601,$3701,$3701
dc.w $3701,$3801,$3801,$3801,$3901,$3901,$3901,$3a01,$3a01
dc.w $3a01,$3b01,$3b01,$3b01,$3c01,$3c01,$3c01,$3d01,$3d01
dc.w $3d01,$3e01,$3e01,$3e01,$3f01,$3f01,$3f01,$4001,$4001
dc.w $4201,$4201,$4401,$4401,$4601,$4601,$4801,$4801,$4b01
dc.w $4b01,$4e01,$4e01,$5101,$5101,$5401,$5401,$6001,$6001
dc.w $7001,$7001,$8001,$8001,$9001,$9001,$a001,$a001,$b001
dc.w $b001,$c001,$c001,$d001,$d001,$e001,$e001
dc.w $e001,$e001,$d001,$d001,$c001,$c001,$b001,$b001,$a001
dc.w $a001,$9001,$9001,$8001,$8001,$7001,$7001,$6001,$6001
dc.w $5401,$5401,$5101,$5101,$4e01,$4e01,$4b01,$4b01,$4801
dc.w $4801,$4601,$4601,$4401,$4401,$4201,$4201,$4001,$4001
dc.w $3f01,$3f01,$3f01,$3e01,$3e01,$3e01,$3d01,$3d01,$3d01
dc.w $3c01,$3c01,$3c01,$3b01,$3b01,$3b01,$3a01,$3a01,$3a01
```

```
dc.w $3901,$3901,$3901,$3801,$3801,$3801,$3701,$3701,$3701
dc.w $3601,$3601,$3601,$3501,$3501,$3501,$3301,$3301,$3301
dc.w $3201,$3201,$3201,$3001,$3001,$3001,$3001
dc.w $ffff
;
; Sprite Daten
;
spritebuffer1: dc.w $a08c,$b000 ;Buffer für Sprite0
sb0:  dc.w $0001,$0000 ;(=Sprite des Spielers)
   dc.w $0007,$0001
   dc.w $001A,$0006
   dc.w $0062,$001E
   dc.w $0184,$007C
   dc.w $0604,$01FC
   dc.w $0808,$07F8
   dc.w $0808,$07F8
   dc.w $1010,$0FF0
   dc.w $1010,$0FF0
   dc.w $2060,$1FE0
   dc.w $2180,$1F80
   dc.w $4600,$3E00
   dc.w $5800,$3800
   dc.w $E000,$6000
   dc.w $8000,$0000
sb1:
   dc.w $0001,$0000
   dc.w $0007,$0001
   dc.w $001A,$0006
   dc.w $0062,$001E
   dc.w $0184,$007C
   dc.w $0604,$01FC
   dc.w $0808,$07F8
   dc.w $0808,$07F8
   dc.w $1010,$0FF0
   dc.w $1010,$0FF0
   dc.w $2060,$1FE0
   dc.w $2180,$1F80
   dc.w $4600,$3E00
   dc.w $5800,$3800
   dc.w $E000,$6000
   dc.w $8000,$0000
sb2:
   dc.w $0000,$0000
   dc.w $0000,$0000
   dc.w $0000,$0000
   dc.w $1FF8,$0008
   dc.w $1008,$0FF8
   dc.w $1008,$0FF8
```

```
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1FF8,$0FF8
  dc.w $0000,$0000
  dc.w $0000,$0000
  dc.w $0000,$0000
sb3:
  dc.w $C000,$4000
  dc.w $7000,$3000
  dc.w $4C00,$3C00
  dc.w $2300,$1F00
  dc.w $20C0,$1FC0
  dc.w $1020,$0FE0
  dc.w $1020,$0FE0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0408,$03F8
  dc.w $0408,$03F8
  dc.w $0304,$00FC
  dc.w $00C4,$003C
  dc.w $0032,$000E
  dc.w $000E,$0002
  dc.w $0003,$0000
;
sprite1: dc.w $a08c,$b000 ;Sprite für Joystick
s1:  dc.w $0001,$0000 ;Phase 1
  dc.w $0007,$0001
  dc.w $001A,$0006
  dc.w $0062,$001E
  dc.w $0184,$007C
  dc.w $0604,$01FC
  dc.w $0808,$07F8
  dc.w $0808,$07F8
  dc.w $1010,$0FF0
  dc.w $1010,$0FF0
  dc.w $2060,$1FE0
  dc.w $2180,$1F80
  dc.w $4600,$3E00
  dc.w $5800,$3800
  dc.w $E000,$6000
  dc.w $8000,$0000
s2:
  dc.w $0000,$0000 ;Phase 2
  dc.w $0000,$0000
```

```
  dc.w $0000,$0000
  dc.w $1FF8,$0008
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1FF8,$0FF8
  dc.w $0000,$0000
  dc.w $0000,$0000
  dc.w $0000,$0000
s3:
  dc.w $C000,$4000 ;Phase 3
  dc.w $7000,$3000
  dc.w $4C00,$3C00
  dc.w $2300,$1F00
  dc.w $20C0,$1FC0
  dc.w $1020,$0FE0
  dc.w $1020,$0FE0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0408,$03F8
  dc.w $0408,$03F8
  dc.w $0304,$00FC
  dc.w $00C4,$003C
  dc.w $0032,$000E
  dc.w $000E,$0002
  dc.w $0003,$0000
;
sprite2: dc.w $a08c,$b000 ;Explosions sprite
ss1:
  dc.w $003E,$003E ;Phase 1
  dc.w $202D,$2037
  dc.w $7C7D,$7C63
  dc.w $37D9,$2FE7
  dc.w $1832,$27CE
  dc.w $6784,$787C
  dc.w $33C4,$3C3C
  dc.w $3A76,$3D8E
  dc.w $1B2A,$1CD6
  dc.w $0BA6,$0C7E
  dc.w $0C7E,$0FFE
  dc.w $087C,$0FFC
  dc.w $1230,$1DF0
  dc.w $17E0,$1860
```

```
  dc.w $1FC0,$1BC0
  dc.w $0E00,$0E00
ss2:
  dc.w $003E,$003E ;Phase 2
  dc.w $3BED,$3BF7
  dc.w $6E7D,$7FE3
  dc.w $6089,$7FF7
  dc.w $58E3,$671F
  dc.w $47D2,$782E
  dc.w $203A,$3FC6
  dc.w $31F7,$3E0F
  dc.w $1FB1,$104F
  dc.w $0BC1,$0C3F
  dc.w $0083,$0F7F
  dc.w $006F,$0FFF
  dc.w $1218,$1DF8
  dc.w $1738,$18F8
  dc.w $1F78,$1BF8
  dc.w $0FC0,$0FC0
ss3:
  dc.w $003E,$003E ;Phase 3
  dc.w $3BED,$3BF7
  dc.w $6E7D,$7FE3
  dc.w $6289,$7FF7
  dc.w $5B63,$679F
  dc.w $435A,$7FA6
  dc.w $583A,$67C6
  dc.w $41CF,$7E77
  dc.w $2DE9,$3677
  dc.w $2B3D,$3CF3
  dc.w $3B55,$3CBB
  dc.w $3359,$3CB7
  dc.w $1352,$1CBE
  dc.w $1624,$19DC
  dc.w $1F6C,$1BFC
  dc.w $0FD0,$0FD0
;
sprite3: dc.w $508c,$6000 ;Feind (roter Punkt)
se0:
  dc.w $03E0,$0020
  dc.w $0C18,$03F8
  dc.w $1004,$0FFC
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
```

```
  dc.w $4001,$3FFF
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $1004,$0FFC
  dc.w $0C18,$03F8
  dc.w $03E0,$01E0
  dc.w $0000,$0000
se1:
  dc.w $03E0,$0020 ;Phase 1
  dc.w $0C18,$03F8
  dc.w $1004,$0FFC
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $4001,$3FFF
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $1004,$0FFC
  dc.w $0C18,$03F8
  dc.w $03E0,$01E0
  dc.w $0000,$0000
se2:
  dc.w $0000,$0000 ;Phase 2
  dc.w $03E0,$0020
  dc.w $0C18,$03F8
  dc.w $1004,$0FFC
  dc.w $1004,$0FFC
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $2002,$1FFE
  dc.w $1004,$0FFC
  dc.w $1004,$0FFC
  dc.w $0C18,$03F8
  dc.w $03E0,$01E0
  dc.w $0000,$0000
  dc.w $0000,$0000
se3:
  dc.w $0000,$0000 ;Phase 3
  dc.w $0000,$0000
  dc.w $03E0,$0020
  dc.w $0410,$03F0
  dc.w $0808,$07F8
  dc.w $1004,$0FFC
```

```
  dc.w $1004,$0FFC
  dc.w $1004,$0FFC
  dc.w $1004,$0FFC
  dc.w $1004,$0FFC
  dc.w $0808,$07F8
  dc.w $0410,$03F0
  dc.w $03E0,$01E0
  dc.w $0000,$0000
  dc.w $0000,$0000
  dc.w $0000,$0000
;
sprite4:
  dc.w $0000,$0000 ;Symbol 1
sm0:
  dc.w $0180,$0080
  dc.w $0240,$01C0
  dc.w $0420,$03E0
  dc.w $0990,$06F0
  dc.w $1248,$0DF8
  dc.w $2424,$1BFC
  dc.w $4812,$37FE
  dc.w $9009,$6FFF
  dc.w $9009,$6FFF
  dc.w $4812,$37FE
  dc.w $2424,$1BFC
  dc.w $1248,$0DF8
  dc.w $0990,$06F0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0180,$0080
sm1:
  dc.w $0180,$0080 ;Phase 1
  dc.w $0240,$01C0
  dc.w $0420,$03E0
  dc.w $0990,$06F0
  dc.w $1248,$0DF8
  dc.w $2424,$1BFC
  dc.w $4812,$37FE
  dc.w $9009,$6FFF
  dc.w $9009,$6FFF
  dc.w $4812,$37FE
  dc.w $2424,$1BFC
  dc.w $1248,$0DF8
  dc.w $0990,$06F0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0180,$0080
sm2:
```

```
  dc.w $0180,$0080 ;Phase 2
  dc.w $0240,$01C0
  dc.w $0420,$03E0
  dc.w $0990,$06F0
  dc.w $1248,$0C78
  dc.w $2424,$183C
  dc.w $4812,$319E
  dc.w $9009,$63CF
  dc.w $9009,$63CF
  dc.w $4812,$319E
  dc.w $2424,$183C
  dc.w $1248,$0C78
  dc.w $0990,$06F0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0180,$0080
sm3:
  dc.w $0180,$0080 ;Phase 3
  dc.w $0240,$01C0
  dc.w $0420,$03E0
  dc.w $0990,$06F0
  dc.w $1248,$0C78
  dc.w $2424,$183C
  dc.w $4812,$301E
  dc.w $9009,$600F
  dc.w $9009,$600F
  dc.w $4812,$301E
  dc.w $2424,$183C
  dc.w $1248,$0C78
  dc.w $0990,$06F0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0180,$0080
;
sprite5:
  dc.w $0000,$0000 ;Symbol 2
sn0:
  dc.w $FFFF,$0001
  dc.w $6002,$1FFE
  dc.w $100C,$0FFC
  dc.w $0810,$07F0
  dc.w $0420,$03E0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
```

```
  dc.w $0420,$03E0
  dc.w $0420,$03E0
  dc.w $1818,$07F8
  dc.w $2004,$1FFC
  dc.w $FFFF,$3FFF
sn1:
  dc.w $FFFF,$0001 ;Phase 1
  dc.w $6002,$1FFE
  dc.w $100C,$0FFC
  dc.w $0810,$07F0
  dc.w $0420,$03E0
  dc.w $0420,$03E0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0240,$01C0
  dc.w $0420,$03E0
  dc.w $0420,$03E0
  dc.w $1818,$07F8
  dc.w $2004,$1FFC
  dc.w $FFFF,$3FFF
sn2:
  dc.w $FFFF,$0001 ;Phase 2
  dc.w $4002,$3FFE
  dc.w $2004,$1FFC
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $0810,$07F0
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $2004,$1FFC
  dc.w $4002,$3FFE
  dc.w $FFFF,$7FFF
sn3:
  dc.w $FFFF,$0001 ;Phase 3
  dc.w $4002,$3FFE
  dc.w $4002,$3FFE
  dc.w $2004,$1FFC
  dc.w $2004,$1FFC
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
```

```
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $1008,$0FF8
  dc.w $2004,$1FFC
  dc.w $2004,$1FFC
  dc.w $4002,$3FFE
  dc.w $FFFF,$7FFF
;
; Copperliste
;
copperl:
       dc.w $008e,$3081 ;DIWSTRT
       dc.w $0090,$35c1 ;DIWSTOP
       dc.w $0104,$0064 ;BPLCON2
       dc.w $0092,$0038 ;DDFSTRT
       dc.w $0094,$00d0 ;DDFSTOP
       dc.w $0102,$0000 ;BPLCON1
       dc.w $0108,$0000 ;BPL1MOD
       dc.w $010a,$0000 ;BPL2MOD
       dc.w $0100,$6600 ;BPLCON0
pl1:   dc.w $00e0,$0000 ;BPL1PTH
       dc.w $00e2,$0000 ;BPL1PTL
pl11:  dc.w $00e4,$0000 ;BPL2PTH
       dc.w $00e6,$0000 ;BPL2PTL
pl2:   dc.w $00e8,$0000 ;BPL3PTH
       dc.w $00ea,$0000 ;BPL3PTL
pl22:  dc.w $00ec,$0000 ;BPL4PTH
       dc.w $00ee,$0000 ;BPL4PTL
pl3:   dc.w $00f0,$0000 ;BPL5PTH
       dc.w $00f2,$0000 ;BPL5PTL
pl33:  dc.w $00f4,$0000 ;BPL6PTH
       dc.w $00f6,$0000 ;BPL6PTL
p1:    dc.w $0120,$0000 ;SPR0PTH
       dc.w $0122,$0000 ;SPR0PTL
       dc.w $0124,$0000 ;SPR1PTH
       dc.w $0126,$0000 ;SPR1PTL
p2:    dc.w $0128,$0000 ;SPR2PTH
       dc.w $012a,$0000 ;SPR2PTL
       dc.w $012c,$0000 ;SPR3PTH
       dc.w $012e,$0000 ;SPR3PTL
p3:    dc.w $0130,$0000 ;SPR4PTH
       dc.w $0132,$0000 ;SPR4PTL
       dc.w $0134,$0000 ;SPR5PTH
       dc.w $0136,$0000 ;SPR5PTL
p4:    dc.w $0138,$0000 ;SPR6PTH
       dc.w $013a,$0000 ;SPR6PTL
       dc.w $013c,$0000 ;SPR7PTH
```

```
       dc.w $013e,$0000 ;SPR7PTL
       dc.w $0180,$0000 ;Farben für kleines
       dc.w $0182,$00ff ;Playfield
       dc.w $0184,$00de
       dc.w $0186,$00cc
       dc.w $0188,$00ab
       dc.w $018a,$009a
       dc.w $018c,$0078
       dc.w $018e,$0067
       dc.w $0190,$0000   ;Farben für großes
       dc.w $0192,$0e84   ;Playfield
       dc.w $0194,$0d73
       dc.w $0196,$0b62
       dc.w $0198,$0a51
       dc.w $019a,$0941
       dc.w $019c,$0730
       dc.w $019e,$0620
sprr0:dc.w $01a0,$0000   ;Farben für Joystick-
       dc.w $01a2,$0f0c   ;Sprite
       dc.w $01a4,$0c0a
       dc.w $01a6,$0a08
sprr1:dc.w $01a8,$0000   ;Farben für Feind-
       dc.w $01aa,$0f00   ;Sprite
       dc.w $01ac,$0b00
       dc.w $01ae,$0800
sprr2:dc.w $01b0,$0000   ;Farben für Symbol-
       dc.w $01b2,$00f0   ;Sprite 1
       dc.w $01b4,$00a0
       dc.w $01b6,$0060
sprr3:dc.w $01b8,$0000   ;Farben für Symbol-
       dc.w $01ba,$0ff0   ;Sprite 2
       dc.w $01bc,$0aa0
       dc.w $01be,$0660
cins:  blk.w 120          ;Platz für 3D-Rolle
       dc.w $ff01,$fffe   ;Wait
       dc.w $0100,$5200   ;BPLCON0
pl111:dc.w $00e0,$0000   ;BPL1PTH
       dc.w $00e2,$0000   ;BPL1PTL
       dc.w $00e4,$0000   ;BPL2PTH
       dc.w $00e6,$0000   ;BPL2PTL
       dc.w $00e8,$0000   ;BPL3PTH
       dc.w $00ea,$0000   ;BPL3PTL
       dc.w $00ec,$0000   ;BPL4PTH
       dc.w $00ee,$0000   ;BPL4PTL
       dc.w $00f0,$0000   ;BPL5PTH
       dc.w $00f2,$0000   ;BPL5PTL
col:   dc.w $0180,$0000   ;Farben für die untere
       dc.w $0182,$00f0   ;Anzeigetafel
```

```
       dc.w $0184,$0fff
       dc.w $0186,$0fff
sieger:dc.w $0188,$0888
       dc.w $018a,$0256
       dc.w $018c,$0888
       dc.w $018e,$0fff
       dc.w $0190,$0267
       dc.w $0192,$0278
       dc.w $0194,$038a
       dc.w $0196,$03ab
       dc.w $0198,$0d33
       dc.w $019a,$0b33
       dc.w $019c,$0a22
       dc.w $019e,$0922
       dc.w $01a0,$0fff
       dc.w $01a2,$0fff
       dc.w $01a4,$0fff
       dc.w $01a6,$0fff
       dc.w $01a8,$0fff
aa1:   dc.w $01aa,$0f77
aa2:   dc.w $01ac,$0c55
aa3:   dc.w $01ae,$0733
       dc.w $01b0,$0f77
       dc.w $01b2,$0e66
       dc.w $01b4,$0d66
       dc.w $01b6,$0c55
       dc.w $01b8,$0a55
       dc.w $01ba,$0944
       dc.w $01bc,$0844
       dc.w $01be,$0733
       dc.w $ffff,$fffe  ;Endekennung
;
;
sg1: =16/8*5
gr1: =320/8*400
gr2: =320/8*600
gr3: =320/8*56
pic1: =blk.b gr1*3
pic2: =blk.b gr2*3
pic3: =blk.b gr3*5
music: =blk.b 74856
life1: =blk.b 500
life2: =blk.b 500
life3: =blk.b 500
finis:
;
```

3. DIE PROGRAMM-DISKETTE

Nicht nur um Ihnen das lästige Abtippen zu ersparen, sondern auch um Druckfehler zu vermeiden, haben wir dem Buch eine Diskette, sowohl mit allen im Buch erklärten Source-Listings, als auch mit den dazugehörigen Grafik- und Musik-Daten, beigelegt. Wenn Sie Ihre Beispieldiskette in df0: einlegen und im CLI mit dem Befehl

```
dir df0: opt a
```

ansehen, sollten darauf folgende Files enthalten sein:

```
Pictures (dir)

  1plane.con                        1Plane.pic
  5planes.con                       5planes.pic

Source (dir)

  BasicScreen.s                     Bild1.s
  Bild2.s                           Blitten.s
  BlitterClr.s                      Copperliste1.s
  Copperliste2.s                    CopperScroll.s
  Flagge.s                          Game.s
  Joystick.s                        Schunkeln.s
  Sprite1.s                         Sprite2.s
  Sprite3.s                         Sprite4.s
  Sprite5.s                         Sternenhimmel.s

Game (dir)

  1.con                             2.con
  3.con                             down.con
  down.pic                          Game
  Game_sprites.pic                  music
  plane_gr.con                      Plane_gr.pic
  plane_kl.con                      plane_kl.pic
```

Auf der Diskette finden Sie alle Daten in den passenden Unterverzeichnissen. In der Lade "Pictures" befinden sich nicht nur die IFF-Bilder, gekennzeichnet durch ".pic", sondern auch die bereits ins RAW-Format konvertierten und somit für die Beispielprogramme verwendbaren Bilder, die mit dem Zeichen ".con" versehen wurden.

Im Unterverzeichnis "Source" sind alle Beispiel-Listings vorhanden, welche mit dem Seka-Assembler erstellt worden sind. Es dürfte aber kein Problem sein, diese auch für andere Assembler zu verwenden.

Im letzten Verzeichnis, "Game", finden Sie das fertige Spiel als Object-code, das bedeutet, Sie können das Programm vom CLI aus aufrufen. Ebenso sind in diesem Verzeichnis alle Bilder und die Musik, die im Spiel selbst verwendet werden.

4. LITERATURVERZEICHNIS

Falls Sie grundlegende Fragen über den Amiga, seine Bausteine oder die Assembler-Programmierung haben, können Sie in der unten angeführten Literatur nachschlagen.

1. Amiga Hardware Reference Manual

   1986 Addison Wesley, Inc.
   ISBN 0-201-11077-6

   update version 1989
   ISBN 0-201-18157-6

2. Eine Einführung in die Assemblerprogrammierung,
   Autor: Nikolaus Laber

   1989 Addison Wesley, Inc.
   ISBN 3-89319-145-3

3. The Amiga DOS Manual, (englische Originalfassung)

   1986 Bantam Books
   ISBN 0-553-34294-0

4. AmigaDOS Handbuch

   Deutsche Übersetztung vom engl. Original, M&T
   ISBN 3-89090-465-3

5. M68000 Familie Teil 1, Hilf/Nausch

   1984 tewi
   ISBN 3-921-803-16-0

6. M68000 Familie Teil 2, Hilf/Nausch

   1984 tewi
   ISBN 3-921-803-30-6

7. Die 68000er Serie Buch 1, Nachmann

1986 Elektor Verlag
ISBN 3-921608-42-2

8. Die 68000er Serie Buch 2, Nachmann

1986 Elektor Verlag
ISBN 3-921608-43-0

9. Amiga Intern, Dittrich/Gelfand/Schemmel

1987 Data Becker
ISBN 3-89090-525-0

5. STICHWORTVERZEICHNIS

VIDEO- UND COMPUTERZENTRUM

Planegger Str. 6/Ecke Am Klostergarten 1, 8000 München 60

Bei uns finden Sie alles unter einem Dach, angefangen von Camcordern, bis hin zu Videorekordern, -nachbearbeitungsgeräten, Mischpulten, Schnittplätze, Genlocks und natürlich den Commodore Amiga mit all seinem Zubehör im Bereich Hard- und Software.



Alle Artikel sind auch im Versand erhältlich.

Wählen Sie die Nummer 089/834 05 91 und geben Sie Ihre Bestellung auf.



***Fachliche Beratung,** durch unsere aus der Presse bekannten Autoren, die Ihnen sicher eine Menge Geld und Ärger sparen wird, steht bei uns an erster Stelle.*

*Unser **Reparaturservice** bringt Ihr defektes Gerät schnell wieder in Schwung.*

Titelservice – Ihr Videofilm wird fernsehreif

Sie senden uns Ihren Videofilm zu und teilen uns mit, an welcher Stelle welche Titel erscheinen sollen – und wir betiteln Ihren Film wunschgemäß.
Eine Videokassette mit einer Menge verschiedener Titeleffekte können Sie gegen eine Schutzgebühr von DM 20,00 anfordern.

Lechner

Verlag Gabriele Lechner
Video- und Computer-Zubehör
Am Klostergarten 1
Ecke Planegger Straße
(2 Minuten vom
Pasinger Marienplatz)
8000 München 60
Telefon 0 89 / 8 34 05 91
Telefax 0 89 / 820 43 55

BESTSELLER AUS DEM VERLAG LECHNER

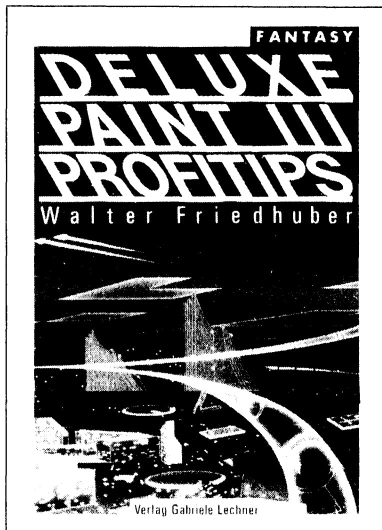
ISBN 3-926858-10-9
340 Seiten,
67 Abbildungen
inklusive 1 Diskette
DM 69,00



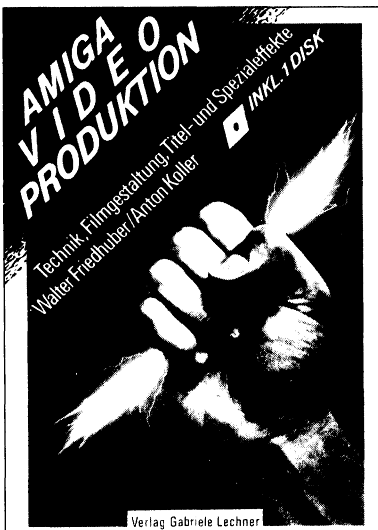
ISBN 3-926858-12-5
360 Seiten,
105 Abbildungen
inklusive 1 Diskette
DM 69,00



ISBN 3-926858-24-9
ca. 450 Seiten,
ca. 150 Abbildungen
inklusive 2 Disketten
DM 98,00



ISBN 3-926858-25-7
550 Seiten
120 Abbildungen
inklusive 1 Diskette
DM 79,00



DISKETTEN ZU DELUXE PAINT



Disk 1: Trickfilm-Elemente

Inhalt: Hintergrundbilder, Anim-Brushes (Explosion, Feuerwerkskörper, galoppierendes Einhorn…) **DM 49,00***



Disk 2: Special Effects

Inhalt: Anim-Brushes (Wellen, Flammen, zerknitterte Coladose, Seifenblasen, Papierblatt im Wind, Skispringer, Abfahrtsläufer, Rennboot, Mississippi-Raddampfer, 3D-Titel und vieles mehr) **DM 49,00***



Disk 3: Tiere

Inhalt: Eine Diskette mit perfekt animierten Tieren, die Sie in Ihr Video einbauen oder in einem Zeichentrickfilm verwenden können. (z.B. Flußpferd, Gepard, Storch, Papagei, Flamingo, usw.) **DM 49,00***



Disk 4: Videofonts

Inhalt: 6 unterschiedliche Schriftsätze in verschiedenen Größen und Muster, davon 1 animierter Font. **DM 49,00***

Alle Preise sind unverbindlich empfohlene Verkaufspreise inkl. MWSt.

SPIELE SELBER PROGRAMMIEREN

Hardwareprogrammierung in Assembler

Spiele faszinierten Menschen schon seit Jahrhunderten und gerade im Zeitalter der Technik, in der die Möglichkeiten zur Spiele-Gestaltung unbegrenzt sind, werden die Computerspiele immer beliebter. Der Amiga eignet sich, durch seine enorme Grafikfähigkeit, hervorragend als Spielekonsole.
Wer schon immer davon geträumt hat, einmal selbst ein Computerspiel zu entwickeln, erhält mit diesem Buch die Grundlage dazu.

Da man nun in Assembler, die optimale „Bewegungsfreiheit" hat, die es ermöglicht kurze und schnelle Routinen zu schreiben, wird in diesem Buch ausschließlich auf diese Sprache eingegangen.

Dem Autor Niki Laber, der bereits eine Reihe professioneller Spiele und Programmpakete entwickelt hat, gelang es auf verständliche Art von der Planung bis zur Programmierung eines kompletten Spieles, alle notwendigen Schritte zu erläutern und anhand von Beispielroutinen aufzuzeigen.
Von der Bilddarstellung bis zur Joystick-Abfrage wird alles besprochen, was zur Spieleprogrammierung notwendig ist.

Sämtliche Listings, sowie zusätzliche Grafiken und Sounds, sind auf der beiliegenden Diskette enthalten.

Verlag Gabriele Lechner
Am Klostergarten 1
8000 München 60

ISBN 3-926858-31-1
DM 69,- SFr 64,- öS 538
inkl. 1 Disk.